Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Interface Web para Execução do Algoritmo CUDAlign para Comparação de Sequências Biológicas em GPU

Jacopo Bellati

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof.ª Dr.ª Alba Cristina Magalhães de Melo

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. Ricardo Zelenovsky

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães de Melo (Orientador) — CIC/UnB
Prof. Dr. Flávio de Barros Vidal — CIC/UnB
Prof. Dr. Li Weigang — CIC/UnB

CIP — Catalogação Internacional na Publicação

Bellati, Jacopo.

Interface Web para Execução do Algoritmo CUDAlign para Comparação de Sequências Biológicas em GPU / Jacopo Bellati. Brasília : UnB, 2014.
71 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. interface web, 2. cudalign, 3. sequências biológicas, 4. gpu

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Interface Web para Execução do Algoritmo CUDAlign para Comparação de Sequências Biológicas em GPU

Jacopo Bellati

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof.ª Dr.ª Alba Cristina Magalhães de Melo (Orientador)
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
CIC/UnB

Prof. Dr. Li Weigang
CIC/UnB

Prof. Dr. Ricardo Zelenovsky
Coordenador do Curso de Engenharia da Computação

Brasília, 13 de dezembro de 2014

Dedicatória

Dedico o presente trabalho a minha família, especialmente a minha mãe, a minha orientadora, a todos as pessoas que cederam seu conhecimento para mim e a meus amigos pessoais.

Dedico, também, aos alunos da UnB da grande área de computação e espero que esse trabalho sirva como fonte de conhecimento e inspiração para vocês.

Agradecimentos

Agradeço a minha mãe, à UnB, ao Departamento de Ciência da Computação da UnB, a minha orientadora, a todos os professores e autores que me passaram conhecimento na área, a meus amigos, colegas e pessoas que, de alguma forma, contribuíram para me tornar o que sou hoje.

Resumo

O CUDAlign é uma ferramenta que se propõe a comparar sequências biológicas em GPU e demanda certos conhecimentos e componentes (tanto de *software* quanto de *hardware*) para ser utilizado com sucesso por um usuário final. Essa demanda torna a ferramenta menos atrativa para o público que não dispõe de tais conhecimentos e componentes.

O presente trabalho de graduação tem por objetivo mitigar o esforço inicial exigido para instalar e utilizar a ferramenta bem como tornar a utilização mais simples e intuitiva. Sendo assim, visa propor e implementar um sistema Web de suporte e monitoramento da execução da ferramenta CUDAlign.

Palavras-chave: interface web, cudalign, sequências biológicas, gpu

Abstract

CUDAlign is a tool that is designed to compare biological sequences in GPU and demands certain knowledge and components (both software and hardware) to be successfully used by end users. This demand makes the tool less attractive to the public that does not possess such knowledge and components.

This graduation project intends to mitigate the initial effort required to install and use the tool and make its use simple and intuitive. Thus, aims to propose and implement a Web system to support and monitor the execution of the CUDAlign tool.

Keywords: web interface, cudalign, biological sequences, gpu

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A comparação de sequências biológicas é uma das operações mais básicas e importantes em Biologia Computacional, sendo amplamente utilizada para determinar o grau de similaridade entre dois organismos [10]. Ao se comparar o material genético de um organismo com outro cujas propriedades já são conhecidas, é possível estimar o nível de similaridade entre as características dos dois organismos [10]. Existem vários algoritmos para realizar a comparação exata de duas sequências biológicas, dentre os quais podemos citar o Needleman-Wunsch [20], Smith-Waterman [24], Gotoh [12] e Myers-Miler [19].

O resultado de uma operação de comparação de sequências pode ser (a) somente o escore, que indica a similaridade entre as mesmas ou (b) o escore e o alinhamento, onde uma sequência (ou parte dela) é colocada sobre a outra (ou parte dela), de maneira a evidenciar as similaridades ou diferenças entre as mesmas.

Os algoritmos existentes que realizam operações de alinhamento demandam alto poder de processamento e memória, fato que é agravado quando a comparação é realizada para sequências muito longas. Por exemplo, o alinhamento de sequências com mais de um milhão de bases pode levar horas ou até mesmo dias e a quantidade de memória necessária pode ser da ordem de *petabytes*. Haja vista a necessidade de algoritmos mais rápidos para obtenção de alinhamentos exatos de sequências biológicas, técnicas de paralelismo são utilizadas para obter um aumento de desempenho no processamento desses alinhamentos.

Dentre as arquiteturas que permitem esse paralelismo, podemos citar as unidades de processamento gráfico (GPUs) [23], que também podem ser utilizadas para executar rotinas não gráficas. Esse tipo de processamento genérico em placa de vídeo é chamado de GPGPU (General-purpose graphics processing units) [8]. Se compararmos o custo-benefício para resolver problemas paralelizáveis, o uso de placas de vídeo pode ser uma boa alternativa quando comparado com a construção de clusters de alto desempenho. Além disso, independente da plataforma utilizada, os algoritmos precisam ser desenvolvidos para não exceder a limitação de memória existente nos ambientes computacionais.

O CUDAlign [8] é uma ferramenta desenvolvida no Programa de Pós-Graduação em Informática da UNB que recupera o alinhamento exato entre duas sequências de DNA longas usando GPU. Atualmente, o CUDAlign é executado via linha de comando, o que o torna pouco atrativo para usuários que não são da área de Ciência da Computação. Além disso, o seu uso é limitado às pessoas que possuem acesso físico às GPUs nas quais o CUDAlign está instalado.

O presente trabalho de graduação tem por objetivo propor e implementar um sistema Web de submissão, suporte e monitoramento da execução da ferramenta CUDAlign. Essa ferramenta se propõe a comparar sequências biológicas em GPU.

Os desafios desse sistema estão em (i) disponibilizar uma interface gráfica amigável para execução do CUDAlign, (ii) obter as sequências a serem comparadas via um sítio na internet e carregá-las na máquina na qual o CUDAlign será executado; (iii) receber os parâmetros fornecidos pelo usuário via Web, montar o *script* de execução do CUDAlign e executá-lo na GPU apropriada; (iv) como cada execução pode demorar horas ou mesmo dias, projetar um meio de monitoramento do andamento da execução; e (v) enviar os dados para o usuário quando a execução terminar.

O restante desse documento está organizado da seguinte maneira. O Capítulo 2 apresenta conceitos e algoritmos importantes para o alinhamento de sequências biológicas. Em seguida, o Capítulo 3 dá uma visão geral sobre o CUDAlign. O Capítulo 4 revisa conceitos importantes sobre a arquitetura de servidores Web. O Capítulo 5 contém o projeto da solução proposta para atender aos requisitos definidos. No Capítulo 6, o fluxo de execução da solução é mostrado sob a ótica do utilizador final, ou seja, o resultado experimental é mostrado. Por fim, no Capítulo 7, conclui-se o trabalho atual e propõem-se trabalhos futuros.

Capítulo 2

Sequências Biológicas

Na biologia, uma sequência é uma ordenação unidimensional de monômeros, covalentemente ligados dentro de um biopolímero, e é, também, referenciada como a estrutura primordial da macromolécula biológica.
Uma sequência biológica representa informação biológica e é representada por um conjunto de letras, ou alfabeto, que indicam sua estrutura e especificam ligações. Uma sequência biológica pode se referir a uma molécula de DNA, RNA ou proteína.
As sequências de DNA são obtidas através do alfabeto $\sum = \{A, T, G, C\}$, RNA através do alfabeto $\sum = \{A, G, C, U\}$ e aminoácidos através do alfabeto $\sum = \{A, R, N, D, E, C, G, Q, H, I, L, K, M, F, P, S, Y, T, W, V\}$.

2.1 Alinhamento e Escore

O alinhamento de sequências biológicas consiste na comparação de duas ou mais sequências a fim de encontrar séries ou padrão de caracteres que estão na mesma ordem. Geralmente, as sequências nas quais são procuradas similaridades são provenientes de proteínas, DNA ou RNA.

O alinhamento de sequências é uma maneira de arranjar sequências biológicas para identificar regiões de similaridade, podendo estas serem relações funcionais, estruturais ou evolucionárias entre as sequências.

Sua classificação é dada de acordo com um escore calculado. Quanto maior o escore, melhor o alinhamento. É preciso, porém, ter cuidado, pois o algoritmo pode mostrar o melhor alinhamento matemático e não, de fato, se o alinhamento faz sentido biológico, ou seja, pode haver falsa similaridade ou falsa correspondência.

Em um alinhamento entre duas sequências, caracteres idênticos (denominados correspondentes ou *matches*) contribuem com um escore positivo. No caso de *mismatches*, ou caracteres que não são idênticos, pode haver a aplicação de um escore positivo, caso haja

alta similaridade entre os caracteres comparados, ou pode haver a aplicação de um escore negativo, caso em que a similaridade é muito baixa. Além disso, há uma terceira possibilidade: certos caracteres, por processos evolucionários, foram extraídos de uma sequência ou adicionados na outra, ou seja, um caracter de uma sequência está alinhado com nenhum caracter da outra sequência. Neste caso, aplica-se um escore negativo conhecido também como penalidade de *gap*.

Existem dois tipos básicos de alinhamento de pares (duas sequências), são eles: alinhamento global e alinhamento local, que serão discutidos a seguir.

Alinhamento global

Este tipo visa obter o melhor alinhamento ponta a ponta, ou seja, todos os caracteres de ambas sequências são utilizados. O propósito é obter o alinhamento global ótimo entre duas sequências. Boas candidatas para a aplicação deste algoritmo são sequências com tamanhos parecidos e similares. O algoritmo de programação dinâmica para resolver esse problema é conhecido como algoritmo de Needleman-Wunsch (Seção 2.4).

Alinhamento local

Este tipo é usado para identificar regiões com alta similaridade. Nesse caso, a procura é por um melhor alinhamento entre subsequências da primeira e segunda sequências, ou melhor alinhamento local. Considerada a melhor abordagem para detectar similaridade entre sequências que possuem muitas divergências. O algoritmo é conhecido como Smith-Waterman (Seção 2.5).

O escore se refere a um termo que, uma vez obtido, classifica um dado alinhamento de sequências biológicas. Esse termo tem como objetivo determinar se o alinhamento em questão aconteceu provavelmente porque as sequências são relacionadas ou por acaso.

Os métodos para cálculo do escore diferem de acordo com o tipo de sequências. Para sequências de DNA e RNA, atribui-se um valor único para *matches*, ou acertos, e um valor único para *mismatches*, ou erros.

A Fig. 2.2 apresenta um alinhamento de DNA e um alinhamento de aminoácidos.

Para sequências de proteínas, um dos métodos mais utilizados para fazer o cálculo do escore entre termos é o log da razão de possibilidades (*log-odds ratio*) e, assim, quanto maior for o escore do alinhamento, melhor o alinhamento. Existe, também, por exemplo, o método dos custos que busca minimizar o escore. O primeiro método é muito utilizado e dele uma matriz de substituição BLOSUM é derivada. Essa matriz, que no caso da Fig. 2.1 é uma matriz 20x20, conhecida, também, por BLOSUM50, denota todos os escores de todos os possíveis pares e é bastante utilizada em algoritmos de comparação de sequências.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**5**	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	**7**	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	**7**	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	**8**	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	**13**	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	**7**	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	**6**	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	**8**	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	**10**	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	**5**	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	**5**	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	**6**	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	**7**	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	**8**	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	**10**	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	**5**	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	**5**	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-4	**15**	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	**8**	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	**5**

Figura 2.1: Matriz de substituição BLOSUM50.

```
 A  T  C  G  -  A          K  I  A  R  N
 T  T  C  G  A  A          R  I  -  T  N
-1 +1 +1 +1 -2 +1 = 1    +3 +5 -2 -1 +7 = 12
```

Figura 2.2: Alinhamento de DNAs e de uma cadeia de aminoácidos.

Quando comparamos sequências, a meta pode ser encontrar evidências de que elas divergiram de um ancestral comum pelo processo de mutação e seleção. O básico do processo de mutação envolve a substituição, inserção ou remoção de resíduos. O número de correspondências (*matches*), substituições e inserções ou remoções - inserções ou remoções atribuem um escore negativo ao alinhamento e, por isso, são conhecidas como penalidades de *gap* - determina o escore do alinhamento. O processo de seleção natural, concebido originalmente por Charles Darwin, envolve a seleção de características desejáveis - que mais contribuem para a sobrevivência do indivíduo no meio - em indivíduos e, por isso, algumas mudanças, ou mutações, podem ser vistas mais que outras.

2.2 Penalidades de buracos

A existência de buracos, quando se trata da comparação entre sequências biológicas, significa que houve uma inserção ou deleção de um monômero em um determinado momento da história evolutiva do ser em análise. Quanto maior o número de buracos em um dado alinhamento, mais distantes as sequências se encontram na cadeia evolutiva.

Existem dois tipos principais de penalidades de *gaps*, ou buracos. São eles: *linear gap* e *affine gap*. O modelo linear, ou *linear gap*, aplica uma penalidade linear proporcional ao tamanho do *gap*. Já o modelo *affine gap* aplica uma penalidade para o *gap* de tamanho um e outra penalidade para os *gaps* imediatamente subsequentes.

2.3 Programação Dinâmica

A ideia por trás da programação dinâmica é quebrar o problema em pequenos subproblemas, resolver esse subproblemas e, ao final, juntar as soluções a fim de obter uma solução global, ou do problema como um todo. No caso específico da otimização do alinhamento de duas sequências biológicas, o escore total é, na verdade, uma soma dos escores entre pares alinhados, ou, em outras palavras, assume-se que cada par alinhado não influencia no fato de outro par estar, também, alinhado.

Esse método é muito usado para o cálculo do alinhamento de duas sequências biológicas pois provê o alinhamento com maior escore, ou ótimo. O método de programação dinâmica engloba as seguintes características segundo [18]:

1. **Alinhamento ótimo**: o método compara cada par de caracteres (um de cada sequência) e produz um alinhamento. Esse alinhamento irá conter acertos, erros e buracos que estarão, nas duas sequências, em posições que irão trazer o maior escore possível. O método foi provado matematicamente e traz o melhor alinhamento, ou maior escore.

2. **Alinhamento global e local**: esses dois casos de alinhamento são passíveis de serem feitos com programação dinâmica apenas aplicando pequenas alterações no algoritmo.

3. **Escolhas**: os alinhamentos obtidos dependem do sistema de escores para comparar os pares de caracteres, como, por exemplo, a matriz de substituição BLOSUM50 (Fig. 2.1) e de como as penalidades de buracos são tratadas.

4. **Alinhamentos, escores de alinhamento e alinhamentos alternativos**: o algoritmo de programação dinâmica produz o escore ótimo, ou maior escore. Existem alinhamentos que possuem dois ou três escores ótimos e existem, ainda, mais alinhamentos que possuem um escore quase tão alto quanto o primeiro. Muitas vezes, biologicamente falando, os escores com taxas de acerto alto, mas não o ótimo, podem ter mais sentido que um alinhamento ótimo. Entender que o algoritmo pode ser facilmente modificado para trazer não só o melhor, mas, também, bons alinhamentos, é importante. Pode haver também os alinhamentos alternativos que são

obtidos calculando-se o alinhamento ótimo, e depois um segundo alinhamento onde as posições do primeiro não se repetem, por exemplo.

5. **Objetivos de análise**: deve-se, ao utilizar tal algoritmo, ter em mente o que se deseja descobrir. O objetivo vai influenciar o modo em que a análise é feita. Escolhas devem ser feitas para chegar à melhor conclusão, são algumas: tipo de programa, alinhamento global ou local, o tipo da matriz de escore e o valor das penalidades de buraco.

2.4 Algoritmo de Needleman-Wunsch

O algoritmo de Needleman-Wunsch [20] é um algoritmo de programação dinâmica e serve para obter alinhamentos ótimos globais com buracos sendo permitidos. A ideia por trás deste algoritmo é fazer uma soma de pedaços independentes e, por isso, funciona.

Dadas, por exemplo, duas sequências de DNA S_0 e S_1 de tamanho m e n, constrói-se uma matriz $\mathbf{H}_{m\times n}$, indexada por i e j, colocando cada monômero das duas sequências como cabeçalhos da matriz somando uma linha $(0, j)$ e uma coluna $(i, 0)$ ao total de linhas e colunas. $\mathbf{H}$ é construída recursivamente inicializando $\mathbf{H}(0, 0)$ com 0 e varrendo a mesma do canto esquerdo superior até o canto direito inferior utilizando a Equação 2.1.

$$\mathbf{H}(i,j) = max \begin{cases} \mathbf{H}(i-1, j-1) + \mathbf{s}(x_i, y_j), \\ \mathbf{H}(i-1, j) - d, \\ \mathbf{H}(i, j-1) - d. \end{cases} \tag{2.1}$$

Nota-se que $\mathbf{H}(i, j)$ é obtido através de uma equação de máximo entre três termos. O primeiro termo representa o alinhamento entre os dois pares e, por isso, soma-se o escore de *match* (se $S(i) = S(j)$) ou *mismatch* (se $S(i) \neq S(j)$) à $\mathbf{H}(i-1, j-1)$. O segundo termo representa que o monômero da sequência colocada como cabeçalho horizontal alinhou-se com um buraco e, por isso, houve aplicação da penalidade d, por fim, o terceiro termo representa que o monômero da sequência colocada como cabeçalho vertical alinhou-se com um buraco, e que também, por isso, houve aplicação da penalidade d.

Enquanto a matriz é construída, guarda-se um ponteiro em cada célula mostrando qual foi a célula que a originou. Esse passo é importante, pois, ao final, essa informação será utilizada para obter o alinhamento ótimo.

Tendo a matriz $\mathbf{H}$ sido construída, para obter o alinhamento ótimo parte-se da última célula, ou a célula do canto inferior direito, e é feito um processo de *traceback*, ou seja, a partir dela a cadeia inteira de células é percorrida utilizando o apontador para a célula de origem discutido anteriormente até que i e j sejam iguais a zero. Existem três casos que

devem ser avaliados quando fazendo este processo e montando o alinhamento, segundo a Fig. 2.3.

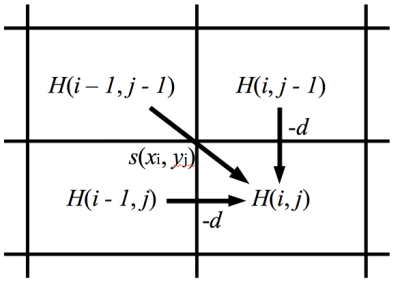


Figura 2.3: Três casos que devem ser avaliados ao montar a matriz de programação dinâmica. Adaptado de [10].

A Fig. 2.4 ilustra a matriz **H** e o alinhamento global entre duas sequências de DNA.

	*	T	A	G	T	C
*	**0**	−2	−4	−6	−8	−10
T	−2	**1**	−1	−3	−5	−7
A	−4	−1	**2**	0	−2	−4
G	−6	−3	0	**3**	**1**	−1
C	−8	−5	−2	1	2	**2**

T	*A*	*G*	*T*	*C*
T	*A*	*G*	–	*C*

Figura 2.4: Exemplo de alinhamento global. [6]

O algoritmo aqui descrito, como se pode ver, recupera apenas um alinhamento ótimo. Existem casos em que pode haver mais de uma derivação em um dado ponto e, nessa situação, é necessário que o algoritmo seja modificado um pouco para recuperar mais de um escore ótimo ou, caso contrário, o mesmo irá escolher arbitrariamente uma das opções disponíveis.

A complexidade desse algoritmo é $O(nm)$, sendo n e m os tamanhos de cada sequência. Como na maioria das vezes os tamanhos são parecidos, ou comparáveis, podemos denotar a complexidade, também, por $O(n^2)$.

2.5 Algoritmo de Smith-Waterman

O algoritmo de Smith-Waterman [24] é usado comumente em situações nas quais estamos procurando o melhor alinhamento local entre duas subsequências. Outro uso, por ser um algoritmo mais sensível que o de alinhamento global, é a busca de similaridade em sequências muito divergentes. O alinhamento com maior escore é chamado de melhor alinhamento local.

Esse algoritmo é muito parecido com o descrito na Seção 2.4, o algoritmo de Needleman-Wunsch, com três diferenças. A primeira diferença é a adição de uma quarta possibilidade à Equação 2.1, obtendo-se a Equação 2.2. Em segundo lugar, o alinhamento pode começar, sendo escolhida a célula com escore mais alto para iniciar o processo de *traceback*, ou acabar, pois o algoritmo considera o encontro de um zero o fim do processo de *traceback*, em qualquer lugar. Finalmente, a primeira linha e coluna da matriz **H** são inicializadas com zero.

$$\mathbf{H}(i,j) = max \begin{cases} 0, \\ \mathbf{H}(i-1,j-1) + \mathbf{s}(x_i, y_j), \\ \mathbf{H}(i-1,j) - d, \\ \mathbf{H}(i,j-1) - d. \end{cases} \tag{2.2}$$

A primeira diferença faz com que um escore que apresente valor abaixo de zero, seja substituído por um zero. A explicação para isso vem do fato de que, se em um dado ponto o alinhamento dos pares possui escore menor que zero, é melhor começar um novo alinhamento ao invés de continuar com o antigo.

A Fig. 2.5 mostra um exemplo de alinhamento local.

	*	T	A	G	T	C
*	**0**	0	0	0	0	0
T	0	**1**	0	0	1	0
A	0	0	**2**	0	0	0
G	0	0	0	**3**	1	0
C	0	0	0	1	2	2

T A G
T A G

Figura 2.5: Exemplo de alinhamento local. [6]

2.6 Algoritmo de alinhamento com *affine gaps*

O algoritmo proposto por Gotoh [12] utiliza uma pontuação para os *gaps* segundo o modelo *affine gap* (Seção 2.2). A probabilidade de ocorrer um *gap*, dado que na comparação do par anterior ocorreu um *gap* é aumentada e, por isso, o algoritmo foi criado a fim de minimizar as penalizações decorrentes desses casos específicos.

A complexidade do algoritmo ainda está na ordem de $O(n^2)$. Apesar disso, ao invés de manter um escore do alinhamento de cada par, é preciso armazenar três. O primeiro,

$\mathbf{H}(i,j)$, representa o melhor escore até (i,j), dado que x_i está alinhado com y_j. $\mathbf{E}(i,j)$ e $\mathbf{F}(i,j)$ representam que x_i e y_j estão alinhados com um *gap* respectivamente.

$$\mathbf{H}(i,j) = max \begin{cases} \mathbf{H}(i-1,j-1) + \mathbf{s}(x_i, y_j), \\ \mathbf{E}(i-1,j-1) + \mathbf{s}(x_i, y_j), \\ \mathbf{F}(i-1,j-1) + \mathbf{s}(x_i, y_j). \end{cases} \tag{2.3}$$

$$\mathbf{E}(i,j) = max \begin{cases} \mathbf{H}(i-1,j) - d, \\ \mathbf{E}(i-1,j) - e. \end{cases} \tag{2.4}$$

$$\mathbf{F}(i,j) = max \begin{cases} \mathbf{H}(i,j-1) - d, \\ \mathbf{F}(i,j-1) - e. \end{cases} \tag{2.5}$$

Como é possível ver pelas Equações 2.3, 2.4 e 2.5, o caso em que um *gap* aparece em uma sequência e, logo em seguida, em outra (caso chamado de inserção seguido de uma deleção), não é contemplado.

2.7 Algoritmo de Hirschberg

O algoritmo de Hirschberg [13] (posteriormente estendido por [19]), que é uma versão do algoritmo de Needleman-Wunsch (Seção 2.4), mantêm a complexidade de tempo $O(nm)$ e muda a complexidade de espaço para $O(n+m)$, sendo n e m o tamanho das sequências. Isso pode não parecer muito para sequências biológicas pequenas com algumas centenas de nucleotídeos mas, para a comparação entre sequências de DNA que geram matrizes com milhões de linhas e colunas, o espaço disponível se torna fator limitante e pode ser solucionado implementando esse algoritmo.

No algoritmo de Hirschberg, o cálculo da matriz gerada em Needleman-Wunsch pode ser feito apenas armazenando duas linhas: a linha atual e a anterior, diminuindo, assim, o espaço de armazenamento necessário neste passo.

Para calcular o alinhamento ótimo de duas sequências, X e Y, e não somente seu escore, deve-se, também, utilizar a estratégia de *dividir para conquistar*, ou seja, em um dado momento, o algoritmo irá dividir a matriz em duas partes: $\mathbf{A}$ e $\mathbf{B}$. O ponto no qual a divisão irá ocorrer será determinado por (u,v), uma célula que pertence ao alinhamento ótimo. Para determinar u é fácil, basta dividir o tamanho da sequência Y por dois e pegar sua parte inteira. Deve-se, agora, encontrar um v que satisfaça a condição de que (u,v) está no caminho do alinhamento ótimo, ou seja, $\mathbf{A}(n/2,(u,v)) + \mathbf{B}(n/2, m-(u,v)) = \mathbf{F}(n,m)$. Uma vez encontrado o v, divide-se a matriz neste ponto e encontram-se os caminhos de $(0,0)$ até $(n/2,(u,v))$ e (n,m) até $(n/2, m-(u,v))$, ou seja, o último indica

que pegamos a sequência reversa. Nota-se que esse procedimento pode ser feito várias vezes até que as sequências e seus alinhamentos tornem-se triviais.

A Fig. 2.6 ilustra o algoritmo de Hirschberg.

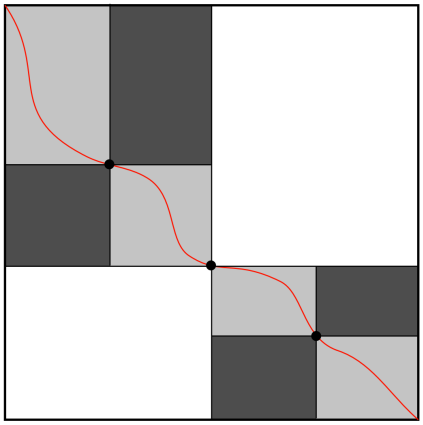


Figura 2.6: Representação do algoritmo de Hirschberg. [9]

Capítulo 3

CUDAlign

Quando uma nova sequência de DNA é descoberta, suas características estruturais e funcionais e informações sobre sua evolução devem ser obtidas. O problema, ao fazer tais comparações, surge quando as sequências possuem tamanhos da ordem de milhões de bases e, consequentemente, requerem um poder de processamento e memória imensos para serem calculados. Nesse contexto, surgiu a proposta do CUDAlign, um algoritmo capaz de trazer um escore ótimo e alinhar (Seção 2.1) utilizando espaço linear, por exemplo, cromossomos inteiros em um tempo razoável.

3.1 Histórico

O CUDAlign, como o próprio nome sugere, foi concebido na plataforma CUDA, que é uma plataforma de processamento paralelo e modelo de programação inventado pela NVIDIA. A plataforma tem sido muito utilizada por desenvolvedores de *software*, cientistas e pesquisadores em diversas áreas para computação em GPU [21].

A primeira versão do CUDAlign, ou CUDAlign 1.0, foi publicada em 2011 e tinha como proposta um algoritmo que calculasse, utilizando GPU, em arquitetura CUDA, em tempo hábil, que utilizasse espaço linear e que não restringisse o tamanho da entrada, o escore ótimo entre duas sequências grandes (com tamanho entre 1 milhão de pares de bases e 47 milhões de pares de bases) de DNA. O cálculo do escore ótimo é feito utilizando Smith-Waterman (Seção 2.5) com *affine gaps* (Seção 2.2). A segunda versão, ou 2.0, foi publicada e estendeu a versão 1.0 para suportar, além do cálculo do escore ótimo, o alinhamento completo ótimo das duas sequências. O alinhamento é obtido com um algoritmo que é a combinação dos algoritmos Myers-Miller (Hirschberg, Seção 2.7) e Smith-Waterman. Essa implementação é limitada apenas pela memória total disponível na GPU e o espaço total disponível em disco. Por fim, uma terceira versão, a 2.1, foi

lançada com uma otimização que melhorou bastante a performance da versão anterior, o processo chamado *block pruning*, que será discutido na Seção 3.3.1.

3.2 CUDAlign 1.0

O CUDAlign 1.0 é uma ferramenta que é capaz de comparar sequências longas de DNA utilizando o algoritmo de Smith-Waterman, a penalidade de *gaps* de *affine gaps* e com memória linear. Sua plataforma de implementação é a arquitetura CUDA, da NVidia. O projeto dessa ferramenta leva em conta diversos fatores que servem para atingir o objeto para o qual foi proposto e para torná-la suficientemente flexível. São eles [9]:

- O limite no tamanho das sequências é determinado apenas pelo tamanho da memória global disponível na GPU.
- A matriz deve ter uma precisão de 32 bits a fim de comportar casos em que o escore de similaridade seja grande. Por exemplo: considerando uma pontuação de *match* +1 e duas sequências idênticas de 100 milhões de pares de bases, o escore gerado seria de 100 mega, o que causaria *overflow* em precisões menores.
- A quantidade de memória utilizada deve estar na ordem de $O(m + n)$, sendo m o tamanho de uma sequência e n o tamanho da outra. Caso a quantidade de memória utilizada estivesse na ordem de $O(mn)$, seria inviável trabalhar com sequências grandes como é o caso do que o algoritmo se propõe a fazer. Por exemplo, a comparação de sequências de 30 milhões de pares de bases resultaria na necessidade de 3.6 petabytes de memória para armazenar a matriz.
- O tempo de execução deve estar na ordem de GCUPS (*Giga Cells Updates per Second*), ou bilhões de células por segundo. Essa medida é um bom referencial de desempenho, pois outras implementações superam 1 GCUPS. Para ilustrar, caso sequências de 10 milhões de pares de bases fossem comparadas com uma velocidade de 10 GCUPS, o tempo necessário para o seu processamento seria menor que 3 horas.
- O tempo de execução deve se manter proporcional ao tamanho das sequências, ou seja, a ferramenta deve ser escalável.
- A ferramenta deve prover um mecanismo de recuperação de execução, ou seja, deve ser possível retomar a execução a partir de um estado previamente salvo, denominado *checkpoint*. Duas vantagens principais podem ser observadas: recuperação de interrupções não programadas (como falta de luz) e parcelamento da execução

Figura 3.1: Diagonais e sentido de execução do *wavefront*. [9]

(executar apenas quando o *hardware* é pouco demandado por outras aplicações, por exemplo).

3.2.1 Paralelismo

O projeto do CUDAlign 1.0 leva em conta o modo como escores são calculados. Nos cálculos realizados pelos algoritmos de Smith-Waterman e Myers-Miller, a maior parte do tempo se gasta calculando as matrizes de programação dinâmica. Observando que o escore da célula $\mathbf{H}(i, j)$ depende das células $\mathbf{H}(i-1, j)$, $\mathbf{H}(i, j-1)$ e $\mathbf{H}(i-1, j-1)$, pode-se perceber que as células que estão em uma mesma diagonal podem ser processadas em paralelo. Geralmente, o paralelismo de fato se dá nessa fase e a técnica utilizada para maximizá-lo é chamada de *wavefront*.

No CUDAlign, houve a divisão das matrizes de programação dinâmica em três níveis lógicos: a própria matriz, que é um aglomerado de blocos; blocos que, por sua vez, são um aglomerado de células; e células, que são a menor parte da matriz, a parte indivisível. Com isso, o paralelismo pode se dar em dois níveis: entre blocos, ou paralelismo externo, e entre células de um bloco, ou paralelismo interno.

Paralelismo externo

A matriz de programação dinâmica é dividida em blocos com R linhas e C colunas cada. A matriz, então, após a divisão, será composta de $m/R \times n/C$ blocos, onde m e n são os tamanhos das sequências. O número B de blocos concorrentes e o número T de *threads* por bloco são os valores que determinam R ($R = \alpha T$, onde α é o número inteiro de linhas que cada *thread* irá executar) e C ($C = n/B$), e são escolhidos de acordo com as especificações da GPU e com resultados empíricos obtidos em cada placa. Após esses cálculos, os blocos são agrupados em diagonais externas e são processados usando a técnica

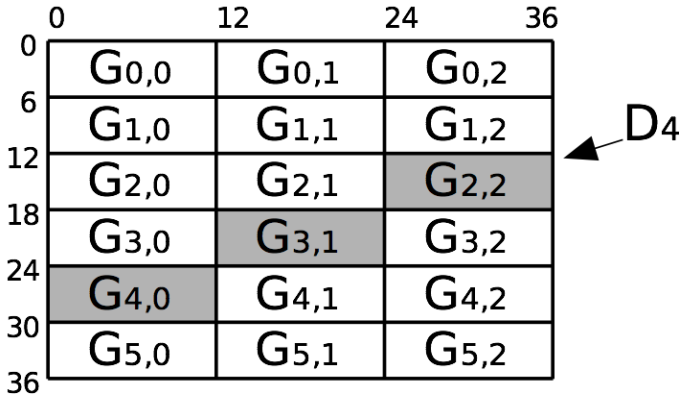


Figura 3.2: Paralelismo externo: diagonal de blocos 4. [9]

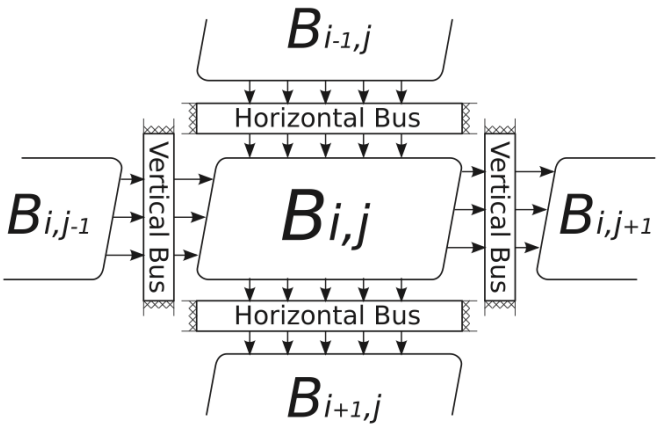


Figura 3.3: Barramento horizontal, que passa a última linha do bloco superior para o bloco inferior, e barramento vertical, que passa os valores das últimas vizinhanças de células do bloco da esquerda para o bloco da direita. [8]

de *wavefront*. Uma diagonal externa só pode ser processada quando a anterior terminou seu processamento. Como é fácil perceber, é necessário que algumas linhas sejam passadas de um bloco superior para um inferior para que os cálculos sejam realizados e, para isso, utiliza-se uma estrutura denominada *barramento horizontal* (cujo tamanho é o próprio tamanho das linhas). Outra estrutura necessária é denominada *barramento vertical*, que serve para passar os valores das últimas vizinhanças de células do bloco da esquerda para o da direita ($T \times B$).

Paralelismo interno

Para cada bloco da matriz, T *threads* processam $R \times C$ células deste bloco. As células são agrupadas em diagonais internas e processadas usando a técnica de *wavefront*. O sentido

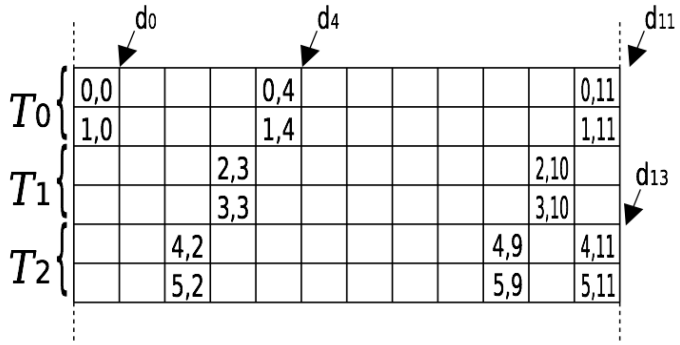


Figura 3.4: Paralelismo interno com $\alpha = 2$. [9]

do processamento de uma *thread* é da esquerda para direita, de cima para baixo. A correta execução do *wavefront* depende das *threads* processarem as células das diagonais internas ao mesmo tempo, sendo, ao final desse processo, sincronizadas, processando a próxima diagonal interna.

3.2.2 Otimizações

Ao propor o algoritmo, diversas otimizações foram implementadas para explorar ao máximo o paralelismo do *hardware*.

Delegação de Células

Como visto anteriormente, o paralelismo externo (Seção 3.2.1) e o paralelismo interno (Seção 3.2.1) utilizam a técnica de *wavefront*. Ao lançar um olhar mais atento à Figura 3.1, vemos que as diagonais do começo e do final possuem menos células para serem processadas paralelamente, ou seja, o potencial de processamento não é utilizado ao seu máximo. Para corrigir este problema e otimizar a execução, uma técnica denominada *delegação de células* foi proposta e o processamento dentro do bloco é feito enquanto o paralelismo da diagonal interna for máximo, deixando, assim, o bloco com um aspecto de paralelogramo. Como é possível ver na Figura 3.5, agora as células riscadas são deixadas como pendentes para o próximo bloco processar e, além disso, o bloco é encarregado de processar as células pendentes de um bloco anterior. As células pendentes são, de fato, as células delegadas.

Divisão de fases

O processo de *delegação de células* (Seção 3.2.2) por si só gera uma falha. O motivo é que o escalonador da GPU pode executar blocos de uma mesma diagonal externa em qualquer ordem e, ao utilizar a técnica anterior, há a formação de uma dependência de dados entre

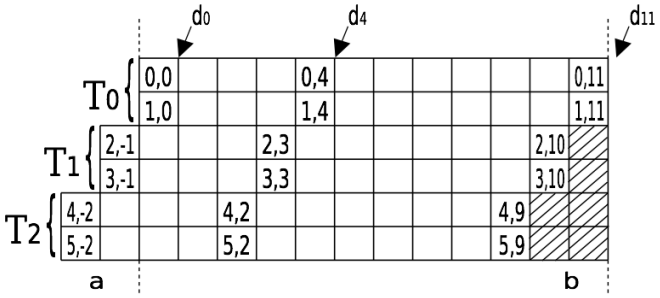


Figura 3.5: Blocos em forma de paralelogramos para aproveitar ao máximo o paralelismo do *wavefront*. [9]

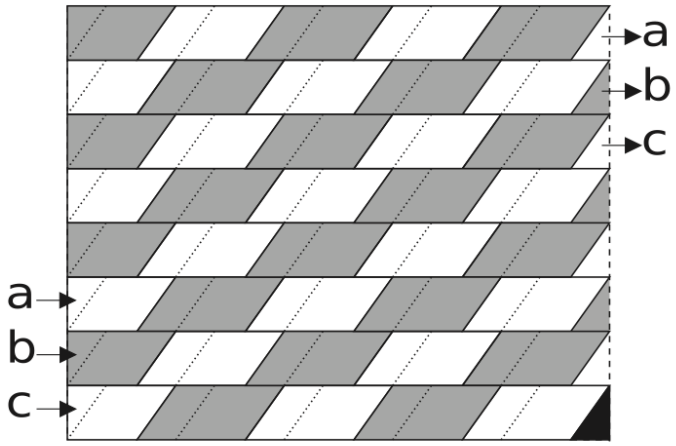


Figura 3.6: Delegação de células e divisão de fases. [8]

os blocos. Para eliminar a dependência entre blocos, a solução é calcular todas as células pendentes antes delas serem lidas, e os blocos devem ser sincronizados para garantir a correta leitura das células. Essa solução, denominada *divisão de fases*, divide a etapa de execução dos blocos em fase curta, que tem como objetivo processar todas as células pendentes, e fase longa, que, uma vez calculadas todas as células pendentes da diagonal externa anterior, termina de calcular as diagonais internas dos blocos. Desse modo, o problema de dependência é sanado, pois cria-se uma forma de sincronismo entre os blocos e a única condição a ser garantida é que o tamanho da fase curta $(T-1)$ seja menor ou igual ao tamanho da fase longa $(n/B-(T-1))$, ou seja, $n \geq 2B(T-1)$, que é a chamada *condição de tamanho mínimo*. Caso esta condição não seja satisfeita, os parâmetros B e T podem ser reduzidos para satisfazê-la.

3.2.3 Tolerância à falhas

A estratégia utilizada pelo CUDAlign para salvar o estado de execução é a de armazenar uma linha completa da matriz de programação dinâmica pois, dessa forma, permite-se que o trabalho possa ser continuado utilizando uma instância do CUDAlign mesmo com um número diferente de blocos ou *threads*, permitindo, inclusive, que diferentes CPUs sejam utilizadas.

3.2.4 Utilização de memória da GPU

A fim de disponibilizar uma máquina para rodar a ferramenta CUDAlign, é necessário conhecer seus requerimentos de memória. A ferramenta armazena, durante sua execução, algumas estruturas importantes em memória e conhecê-las é um passo importante para mensurar corretamente os requisitos do sistema.

Para sequências muito grandes, uma aproximação confiável da quantidade de memória de GPU utilizada seria $9n + m$ bytes. Na Figura 3.8, encontram-se as estruturas discriminadas segundo seu nome, tamanho necessário e tipo.

3.3 CUDAlign 2.1

O algoritmo proposto foi implementado em CUDA, C++ e *pthreads*.

O CUDAlign versão 2.1 recebe duas sequências de entrada, podendo cada uma dessas ter milhões de bases, e, como resultado, produz o escore ótimo e alinhamento local total em tempo e espaço linear. A ideia principal é obter algumas coordenadas do alinhamento ótimo e crescer o seu número iterativamente até que seja possível recuperar o alinhamento completo usando uma quantidade razoável de memória. O algoritmo executa em seis estágios, são eles:

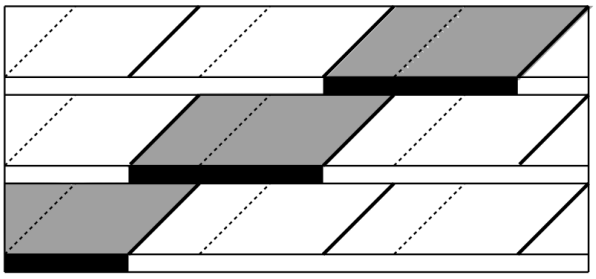


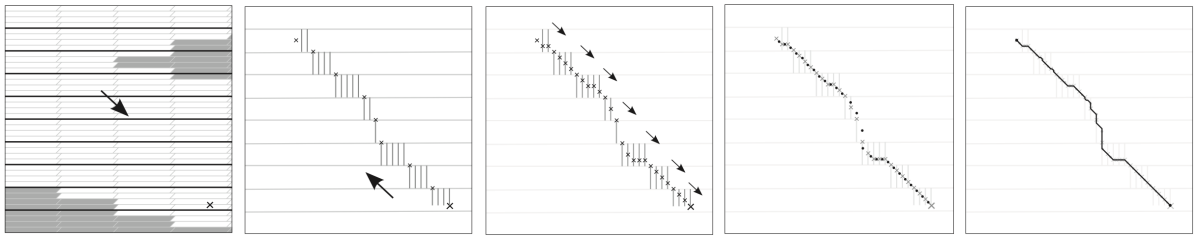
Figura 3.7: Barramento Horizontal dividido em várias linhas da matriz. Para salvar uma linha completa da matriz são necessárias várias iterações. [9]

Estrutura	Tamanho	Tipo
Sequências	$m + n + 2 \times B \times T$ *bytes*	Textura
Vizinhança-K	$(3 \times \alpha + 3)$ *registradores*	Registradores
Memória Compartilhada	$16 \times T$ *bytes*	Memória Compartilhada
Barramento Horizontal	$8 \times n$ *bytes*	Memória Global + Textura
Barramento Vertical	$(8 \times \alpha + 8) \times B \times T$ *bytes*	Memória Global

Figura 3.8: Utilização de memória em GPU do CUDAlign 1.0 discriminada. [9]

- **Estágio 1**: CUDAlign 1.0 (Seção 3.2): calcula as posições nas matrizes de programação dinâmica e salva algumas linhas em disco. Esse estágio é executado em GPU.
- **Estágios 2, 3 e 4**: encontra diversos pontos que pertencem ao alinhamento ótimo em *linhas* e *colunas especiais*. O objetivo desses estágios é quebrar o problema em subproblemas (espaço entre os pontos), seguindo uma abordagem *dividir-para-conquistar*. Os estágios 2 e 3 rodam em GPU enquanto o estágio 4 em CPU.
- **Estágio 5**: calcula o alinhamento ótimo de todos os subproblemas gerados no estágio anterior e, ao final, consolida-os para gerar o alinhamento local ótimo total. Estágio feito em CPU.
- **Estágio 6**: disponibiliza o resultado para visualização. Estágio processado em CPU.

A Figura 3.9 ilustra os estágios 1-5 do CUDAlign.



(a) Estágio 1: blocos que não serão calculados em cinza e *linhas especiais* em negrito

(b) Estágio 2: as primeiras partições e o sentido de execução

(c) Estágio 3: *pontos-chave* e sentido de execução

(d) Estágio 4: muitos *pontos-chave* determinados

(e) Estágio 5: alinhamento completo

Figura 3.9: Estágios de execução do CUDAlign. [8]

3.3.1 Estágio 1

O estágio 1 da execução do CUDAlign versão 2.1 é basicamente o CUDAlign versão 1 inteiro (Seção 3.2). As diferenças são duas.

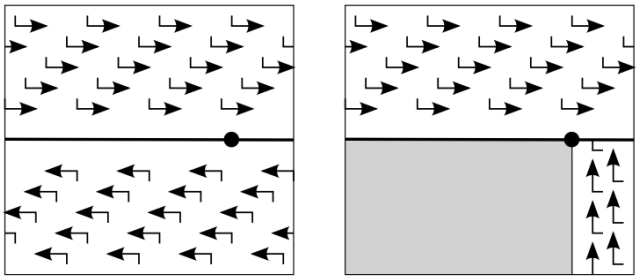


Figura 3.10: Ilustração da *execução ortogonal*. [8]

Primeiro, algumas linhas são salvas em disco (*linhas especiais*). O objetivo desse passo é, além de acelerar a execução de estágios futuros, salvando tempo computacional do cálculo do algoritmo original de Myers-Miller, prover um modo de recuperar a execução caso haja alguma interrupção. As linhas salvas são as últimas de cada bloco, ou seja, as que estão no *barramento horizontal* (Seção 3.2.1). Nota-se que as únicas candidatas para salvamento, então, são as que estão em uma posição múltipla da altura do bloco (αT).

Segundo, uma nova otimização, denominada *block pruning* foi proposta. O objetivo deste passo é eliminar o cálculo de blocos de células que não pertencem ao alinhamento ótimo com certeza. A ideia é fornecer uma prova de que não é matematicamente possível esses blocos produzirem um escore maior do que o que já se tem. O interessante desse método é que, diferentemente dos outros que testam cada célula e o escore ótimo já é conhecido, este testa o bloco inteiro de uma só vez sem conhecer o alinhamento ótimo, sendo, assim, sua implementação viável computacionalmente e trazendo benefícios de tempo.

3.3.2 Estágio 2

O objetivo do estágio 2 é encontrar *pontos-chave*, ou seja, pontos na matriz que fazem parte do alinhamento ótimo, nas *linhas especiais* que foram salvas no estágio 1 e obter o ponto inicial do alinhamento. Esse passo começa do ponto final do alinhamento ótimo encontrado no estágio 1, executando um alinhamento semiglobal, na direção reversa e em GPU. Durante a execução desse estágio, algumas *colunas especiais* são salvas em disco.

O estágio 2 computa apenas as células perto do alinhamento global sem usar a otimização *block pruning*. O processamento, no entanto, ocorre usando, ainda, a *delegação de células* (Seção 3.2.2) e a *divisão de fases* (Seção 3.2.2) mas as constantes B e T podem ser diferentes bastando, apenas, que a *condição de tamanho mínimo*, cujo tamanho considerado agora não é o tamanho de uma sequência (n) e sim a distância entre duas linhas especiais, seja contemplada.

Duas otimizações são introduzidas nesse estágio, são elas: *procedimento de matching baseado em um objetivo* e *execução ortogonal*. A primeira, devido ao fato de o escore máximo, ou *escore-objetivo*, ser conhecido, permite parar o algoritmo assim que o escore máximo é encontrado. Para tirar proveito da primeira otimização, a execução das *threads* é feita verticalmente (*execução ortogonal*, conforme Figura 3.10, e seu objetivo é diminuir a área processada até que correspondência ocorra. [7]

3.3.3 Estágio 3

O objetivo do estágio 3 é obter mais *pontos-chave*. O estágio 3 e o estágio 2 são bastante parecidos e a diferença principal é que o estágio 3 possui partições com início e fim definidos. A ordem em que as partições são executadas não importa e, por isso, elas podem serem processadas em paralelo perfeitamente.

Nesse estágio, a execução das *threads* volta a ser igual à do estágio 1, ou seja, horizontal. Outra informação importante é que as constantes T e B podem mudar novamente e devem, ainda, satisfazer a *condição de tamanho mínimo*.

3.3.4 Estágio 4

O objetivo do estágio 4 é calcular tantos *pontos-chave* quantos forem necessários para alcançar a condição de *tamanho máximo da partição*, ou distância entre sucessivos pares de *pontos-chave*. Esse estágio executa o algoritmo de Myers-Miller entre *pontos-chave*, em CPU, usando múltiplas *threads*. Como no estágio 3, a ordem de processamento é irrelevante, ou seja, há a possibilidade de o processamento ser feito em paralelo.

A otimização proposta nesse estágio chama-se *divisão balanceada* e tem por objetivo diminuir o número de iterações desse estágio. Como visto na Seção 2.7, em Myers-Miller uma partição é dividida ao meio na linha do meio. No caso, essa otimização divide a partição em sua maior dimensão, ou seja, pode dividir tanto com a coluna do meio quanto com a linha do meio. Isso garante que a maior dimensão seja sempre reduzida em cada iteração.

3.3.5 Estágio 5

O *output* deste estágio é o alinhamento ótimo total guardado em disco. Cada partição é alinhada em CPU e todos os resultados são concatenados.

A fim de diminuir o espaço necessário para guardar o *output*, foi desenvolvido um método para representar o alinhamento sem a necessidade de salvar a sequência de caracteres.

3.3.6 Estágio 6

O estágio 6, que é um estágio tido como opcional, tem como objetivo a visualização do alinhamento. O estágio reconstrói o alinhamento a partir do *output* do estágio 5.

Capítulo 4

Arquitetura de Servidores Web

Aplicações desenvolvidas para a Web se comunicam entre si pela rede utilizando um protocolo de comunicação HTTP (*Hypertext Transfer Protocol*). Talvez, o exemplo mais próximo do leitor e, no mundo de hoje, abundante de aplicações Web sejam os *sites*, ou sítios. A comunicação, nesse caso, ocorre, normalmente, entre um *browser* cliente, executando o chamado de lado cliente do HTTP, e um servidor Web, que implementa o lado servidor HTTP. O interessante da arquitetura Web é que diversos ambientes de clientes de uma aplicação podem acessar o mesmo conteúdo sem que o lado servidor necessite de alguma mudança para se adequar a cada um deles, bastando, nesse caso, que as aplicações do lado cliente e o servidor tenham implementado corretamente as definições da RFC (*Request For Comments*) HTTP.

4.1 Terminologia

RFCs contêm notas técnicas e organizacionais acerca da *internet*. Elas cobrem muitos aspectos de redes de computadores, incluindo protocolos, procedimentos, programas, conceitos, etc [1]. Uma página Web é formada por objetos. Nesse contexto, um objeto é um arquivo (que pode ser de qualquer tipo) e é endereçável e acessível por uma URL (*Uniform Resource Locator* ou, em português, Localizador Uniforme de Recurso) única. A URL, particularmente quando usada em conjunto com o HTTP, é conhecida por endereço Web. Na maioria das páginas Web, existe um objeto base, denominado arquivo HTML (*HyperText Markup Language*, linguagem de marcação padrão para criação de páginas Web) base que, utilizando a URL, faz referência a outros objetos tais como imagens, vídeos, *applets*, arquivos HTML etc. Cada URL possui dois componentes: nome do hospedeiro da página e o caminho do objeto. Por exemplo: `http://www.unb.br/img/panoramica/img-home1.jpg` possui `www.unb.br` como nome do hospedeiro da página e `/img/panoramica/img-home1.jpg` como caminho do objeto.

Web *browsers*, tais como como o Internet Explorer, Google Chrome ou FireFox, são aplicações cliente/servidor que utilizam o protocolo HTTP e implementam o lado cliente do HTTP. Por esse motivo, a palavra cliente e Web *browser* poderão ser usadas com o mesmo sentido ao longo do texto. Servidores Web implementam o lado servidor do HTTP e armazenam objetos Web, cada um endereçável por uma URL.

4.2 Arquitetura Cliente/Servidor

Na comunicação entre um par de processos, [25] e [17] definem processos como:

- **Cliente**: o processo que inicia a comunicação, ou seja, que inicialmente contacta o outro processo no começo da sessão.
- **Servidor**: o processo que espera para ser contactado no começo da sessão.

Em um contexto Web, a arquitetura conta com um servidor Web sempre ligado, com endereço fixo conhecido e que serve objetos Web para, potencialmente, milhões de *Web browsers* diferentes [17]. Em outras palavras, teoricamente é possível acessar e utilizar um serviço Web de qualquer lugar do planeta utilizando apenas um dispositivo capacitado e o endereço onde o próprio serviço está hospedado. Importante notar que os clientes não se comunicam diretamente, apenas por intermédio do servidor Web [17].

Como o lado cliente e o lado servidor, na grande maioria dos casos, não compartilham memória, isso dificulta bastante a utilização de métodos de comunicação comuns a sistemas operacionais (filas de mensagens, semáforos, monitores, etc.) [25]. Portanto, a troca de mensagens é utilizada para estabelecer a comunicação entre cliente e servidor.

A troca de mensagens é feita por meio de um protocolo, ou um conjunto de regras e passos, para garantir que as mensagens enviadas, tanto no sentido cliente para servidor e servidor para cliente, sejam corretamente escritas e lidas, ou seja, que a comunicação seja possível.

Além disso, em uma arquitetura cliente/servidor, deve-se levar em conta o canal de comunicação. Normalmente, em um contexto Web, o processo cliente e o processo servidor estão separados fisicamente, com uma distância considerável entre os dois. A fim de se estabelecer essa conexão, um processo envia uma mensagem pela rede por uma interface de *software* denominada *sockets*. O processo assume que há uma infraestrutura de transporte que irá enviar a mensagem do remetente até o destinatário com segurança. Na internet, os processos remetente e destinatário são identificados pelo endereço IP e uma porta associada. A porta 80, por exemplo, identifica um servidor Web.

Exemplos de aplicações que utilizam uma arquitetura cliente/servidor são: Web, FTP (*File Transfer Protocol*), *Telnet* e correio eletrônico.

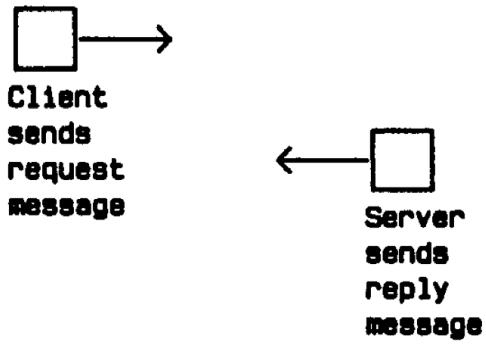


Figura 4.1: Exemplo de interação entre cliente e servidor [25]

Como visto na Figura 4.1, a comunicação é originada do lado cliente que faz um pedido ao servidor. O servidor recebe o pedido, processa-o e devolve uma resposta ao pedido. As primitivas que governam essa interação padrão podem ser [25]:

Confiáveis ou não-confiáveis
: Primitivas confiáveis apresentam garantia de entrega da mensagem. Normalmente é empregado um *acknowledgement* (ou ACK), que indica que uma mensagem foi reconhecida ou recebida com sucesso. O ACK é enviado pelo receptor.

Bloqueadas ou não-bloqueadas
: Primitivas bloqueadas fazem com que um dos lados da comunicação fique com a execução suspensa (bloqueado) até que a resposta a sua solicitação chegue.

Bufferizado* ou não-*bufferizado
: Primitivas *bufferizadas* armazenam mensagens ainda não atendidas. Por outro lado, primitivas não-*bufferizadas* descartam essas mensagens, havendo necessidade de retransmissão.

4.3 Protocolo Web

O protocolo utilizado para comunicação entre aplicações Web é o HTTP (RFCs 1945 [14], para versão 1.0, e 2616 [15], para versão 1.1) e o mesmo apresenta um conjunto de regras que devem ser seguidas para que diferentes processos de uma aplicação Web, executando em diferentes sistemas finais, possam se comunicar passando mensagens uns para os outros. Um protocolo desses define [17]:

- O tipo das mensagens trocadas, como por exemplo mensagens de pedido ou resposta.

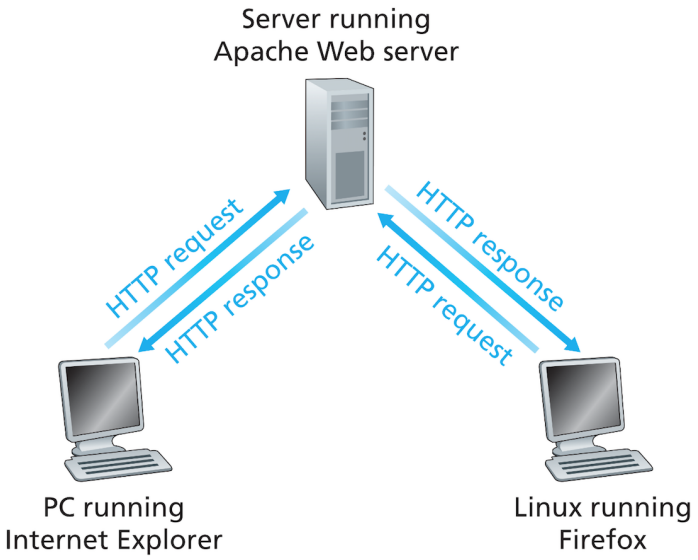


Figura 4.2: Servidor atendendo à requisições Web. [17]

- A sintaxe de vários tipos de mensagens. Define os campos na mensagem, seus tamanhos e o que carregam.
- A semântica dos campos, ou seja, qual é a informação que está presente nos campos.
- Regras de como um processo determina quando e como um processo envia e responde mensagens.

O HTTP é implementado em dois programas distintos na Web: o programa cliente e o programa servidor. Em uma perspectiva focada em *sites*, o HTTP define como *Web browsers* requisitam páginas Web de servidores Web e como servidores transferem páginas Web para clientes. O programa cliente e o programa servidor, executando em diferentes sistemas, se comunicam trocando mensagens HTTP. O HTTP define a estrutura dessas mensagens e como o cliente e servidor trocam mensagens.

Seguindo a pilha TCP/IP, o HTTP, um protocolo da camada de aplicação, faz uso do protocolo da camada de transporte, o TCP (*Transmission Control Protocol*). O TCP é um protocolo que transfere as informações de maneira confiável, já que garante a entrega ao destinatário. [17]

Um fato importante de ser notado é que o protocolo não armazena informação sobre o cliente, ou seja, caso um cliente peça a mesma informação duas vezes em um intervalo de alguns segundos, o servidor não irá acusar que já enviou o arquivo. Ao invés disso, irá reenviar o que foi pedido. [17]

A Figura 4.2 mostra a interação entre um servidor Web e dois clientes diferentes executando em sistemas operacionais diferentes.

4.3.1 HTTP não-persistente vs. HTTP persistente

O modo padrão de operação do HTTP é com conexões persistentes e *pipelining*. Existe, também, no HTTP, o modo de operação com conexões não-persistentes.

No modo conexão persistente, o servidor, após enviar uma resposta, mantém a conexão aberta independente do número de pedidos que recebe. A conexão só é fechada quando um temporizador de inatividade estoura. A vantagem desse método é a diminuição do *overhead* causado pela abertura de diversas conexões TCP e o poder de fazer o chamado *pipelining*, ou seja, fazer o pedido de novos objetos sem que os objetos pendentes tenham necessariamente chegado.

No modo conexão não persistente, ao receber o pedido de um cliente, o servidor devolve uma resposta e fecha a conexão (que só é fechada de fato ao chegar o ACK do cliente). Caso a resposta do servidor necessite que o cliente solicite mais objetos, para cada requisição de objeto haverá a abertura de uma nova conexão TCP. Os *Web browsers* mais modernos oferecem a opção de paralelizar essas conexões, mas mesmo assim o *overhead* causado é muito grande. É importante notar que novos pedidos por objetos são feitos apenas quando a resposta para objetos pendentes chega.

4.3.2 Formato de mensagens HTTP

As RFCs 1945 e 2616 que condizem com a especificação do HTTP, definem, dentre outras coisas, os formatos tanto das mensagens de requisição quanto de resposta. Respeitar o formato descrito é imprescindível para que a comunicação possa ser bem-sucedida entre aplicações que utilizam esse protocolo.

Mensagem de requisição HTTP

A mensagem de requisição HTTP versão 1.1 identifica, dentre outros, informações relevantes para atender à requisição de forma correta e obter o resultado desejado pelo cliente. A Figura 4.3 ilustra o formato da mensagem de requisição.

Uma breve descrição das estruturas e campos importantes de uma mensagem de requisição HTTP se encontra a seguir:

Request Line

Possui os campos *method*, URL e *version*. O campo *method* será discutido futuramente. O campo URL, apesar do nome, carrega, na maioria das vezes, salvo na utilização de *proxies*, apenas o caminho do objeto 4.1 [15] e identifica unicamente um arquivo ou ação dentro deste contexto específico e, por fim, o campo *version* identifica a versão utilizada do protocolo HTTP.

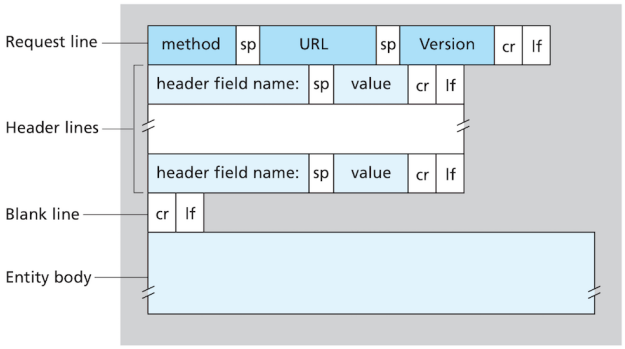


Figura 4.3: Formato padrão de uma requisição HTTP. [17]

Header Lines

O campo *Header Lines* é utilizado para passar informações adicionais sobre a requisição, e sobre o próprio cliente, para o servidor. Esses campos agem como modificadores da requisição [14]. Para exemplificar, um cliente pode informar explicitamente quais os formatos que aceita na resposta.

Entity Body

Usado para enviar dados diversos. Uma *entity*, ou entidade, pode ser um arquivo ou informações enviadas do cliente para o servidor (mensagens de requisição HTTP, utilizando o método *PUT* ou *POST*). Quando presente, pode vir associado a um *Entity Header Line* que fornece metadados sobre o *Entity Body*, ou seja, dados sobre, por exemplo, o tamanho e formato do conteúdo.

Cada mensagem de requisição, como foi visto, possui um campo *method*, ou *método*. O campo método atribui um significado, acima de tudo, sintático à mensagem, além de restringir certas ações com base na escolha de qual será usado. A maioria dos métodos comuns do HTTP versão 1.1 será discutida abaixo de acordo com [15].

OPTIONS

Representa uma requisição de informações sobre as opções de comunicação disponíveis na cadeira de requisição/resposta identificada pelo campo URL no *Request Line*. Quando o campo URL é um asterisco ("*"), a requisição *OPTIONS* é aplicada ao servidor como um todo, mas como, normalmente, as opções dependem do recurso sendo acessado, serve, na verdade, como um *ping*. Esse método é idempotente, ou seja, o efeito de fazer uma ou várias requisições seguidas com esse método é o mesmo.

GET

O método *GET* existe para acessar qualquer informação identificada pelo campo

URL. Indica, também, que a informação deve ser retornada na forma de uma *entity*, ou entidade. Este método é dito seguro, pois, por convenção, não deve tomar nenhuma ação que não seja recuperar informação, e idempotente.

HEAD

O método *HEAD* é similar ao *GET*. A diferença é que, ao receber uma requisição desse tipo, o servidor não deve retornar nada no corpo da mensagem. É um método interessante para, por exemplo, obter metadados sobre a entidade identificada pelo campo URL. *HEAD* é um método seguro e idempotente.

POST

O método *POST* é usado para pedir ao servidor que aceite a entidade presente na requisição como subordinada (assim como um arquivo pertence a um diretório) do recurso identificado pelo campo URL, ou seja, identifica quem no servidor gerenciará a entidade enviada. A ação do método é determinada pelo servidor e depende, geralmente, do recurso sendo acessado.

PUT

O servidor, ao receber uma requisição *PUT*, deve assumir que a entidade enviada na requisição é o próprio recurso identificado pelo campo URL. Se o recurso no campo URL não existir e o servidor for capaz, o servidor pode criar o recurso e este deve ser acessível pelo campo URL. Caso o recurso já exista, o servidor deve considerar a entidade como a nova versão do recurso e substituí-la. É um método idempotente.

DELETE

O método *DELETE* é usado para pedir ao servidor que delete um recurso identificado pelo campo URL. *DELETE* é um método idempotente.

TRACE

Ao enviar uma mensagem de requisição do tipo *TRACE*, o servidor deve retornar para o cliente a mesma mensagem recebida como entidade. Esse tipo de requisição ajuda a testar ou diagnosticar informação. *TRACE* é um método idempotente.

Mensagem de resposta HTTP

A mensagem de resposta HTTP versão 1.1 reporta ao cliente se a requisição dele foi bem-sucedida ou não, uma frase de sucesso ou erro associada à requisição e outros dados. A Figura 4.4 ilustra o formato da resposta HTTP.

Uma breve descrição das estruturas e campos importantes se encontra a seguir:

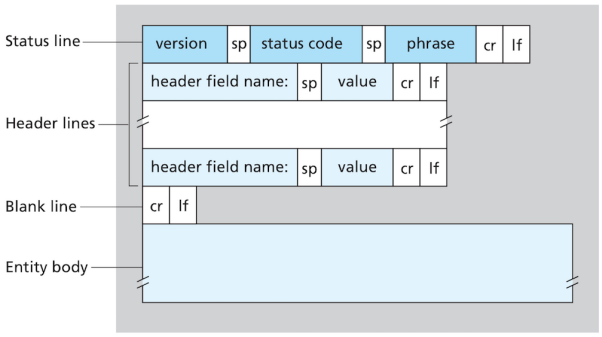


Figura 4.4: Formato padrão de uma resposta HTTP. [17]

Status Line

Possui os campos *version*, *status code* e *phrase*. O primeiro indica qual a versão do protocolo HTTP utilizada. O segundo apresenta um código numérico que representa o resultado da requisição, como, por exemplo, o código 404. Por fim, o terceiro campo carrega uma mensagem associada ao código do campo *status code* que, no caso do exemplo anterior, seria, por padrão, "Não encontrado".

Header Lines

O campo *Header Line* é utilizado para passar informações adicionais sobre a resposta que não puderam ser colocadas na *Status Line*. Esses campos dão informações sobre o servidor e sobre o acesso ao recurso identificado pelo campo URL da requisição [15] (Seção 4.3.2).

Entity Body

Usado para enviar dados diversos. Uma *entity*, ou entidade, pode ser um arquivo ou informações enviadas do servidor para o cliente. Quando presente, pode vir associado a um *Entity Header Line* que fornece metadados sobre o *Entity Body*, ou seja, dados sobre, por exemplo, o tamanho e formato do conteúdo.

O campo *status code* em *Status Line* tem alguns valores padrão definidos na RFC 2616. Ao todo, são quatro classes de mensagens de estado, são elas, de acordo com [15]:

1xx

Códigos de estado que estão na centena que inicia com o número 1 representam uma resposta provisória. Por exemplo: informar ao cliente para continuar com a requisição.

2xx

Códigos de estado que estão na centena que inicia com o número 2 representam que

a requisição que gerou a resposta foi feita com sucesso. Por exemplo: informar que tudo ocorreu bem, que um recurso foi criado com sucesso, etc.

3xx

Códigos de estado que estão na centena que inicia com o número 3 indicam que o cliente deve tomar mais medidas para completar a requisição. Por exemplo: informar que o conteúdo que está querendo acessar foi mudado permanentemente de local.

4xx

Códigos de estado que estão na centena que inicia com o número 4 representam um erro que ocorreu ao tentar processar a requisição. Por exemplo: informar que o conteúdo que está querendo acessar não foi encontrado.

4.3.3 Interação com o usuário: *Cookies*

Existem situações em que armazenar o estado do usuário é interessante, seja para autenticá-lo, rastreá-lo no sentido de saber o que ele faz no sítio ou apenas identificá-lo a fim de mostrar algum conteúdo diferenciado. Como já dito, o HTTP não armazena estado e, nesse contexto, sendo uma tecnologia interessante, surgiram os denominados *Cookies*, que são arquivos texto que armazenam alguma informação e são administrados pelo *Web browser* do cliente. A tecnologia dos *Cookies* é composta de quatro elementos básicos, são eles [17]:

1. **Uma linha de cabeçalho na mensagem de resposta HTTP**: requisita a criação de um Cookie na máquina cliente.
2. **Uma linha de cabeçalho na mensagem de requisição HTTP**: envia o Cookie existente para o sítio.
3. **O arquivo *Cookie***: presente na máquina do cliente, gerenciado pelo *browser* do cliente e contém as informações de que o servidor pediu gravação.
4. **Banco de dados**: presente na máquina do servidor e serve para associar um usuário, normalmente, a um *Cookie*.

A Figura 4.5 ilustra o uso de *Cookies*. Inicialmente, o cliente, buscando acessar um conteúdo na Web, envia uma requisição ao servidor. Como, em um primeiro momento, a requisição não veio com o número do *Cookie* no *Header Line*, o servidor assume que um novo usuário entrou no *site*, cria uma entrada para ele no banco de dados e, ao devolver a página requisitada via mensagem de resposta HTTP, envia, junto com a mensagem, no *Header Line*, o modificador *Set-cookie* com o identificador único do usuário. Ao receber a resposta, o *Web browser* armazena um arquivo texto com a informação necessária e o

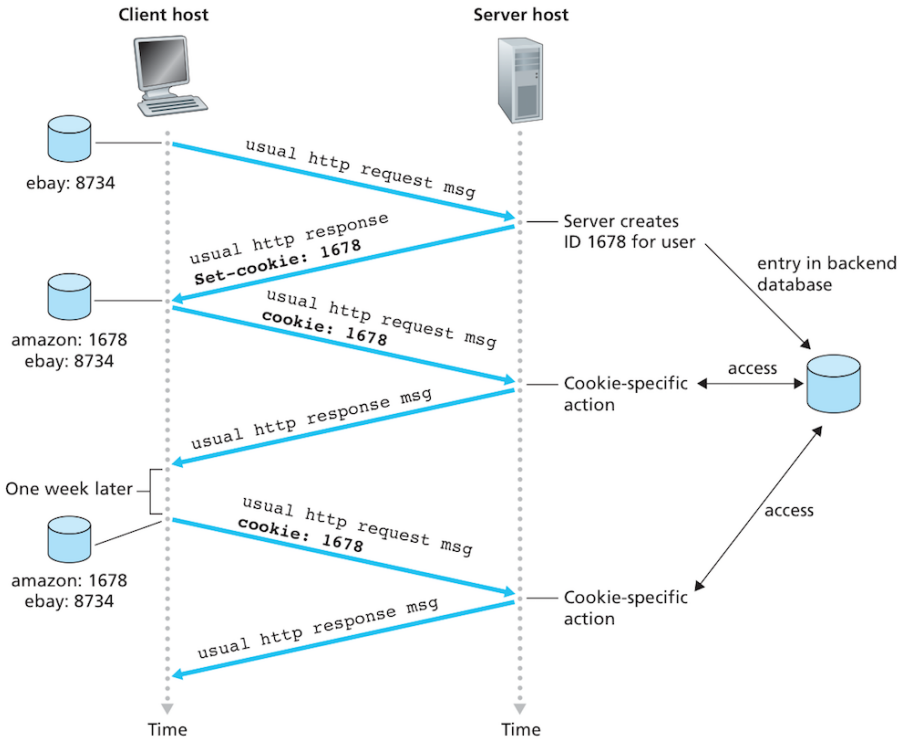


Figura 4.5: Esquema demonstrando o uso de Cookies. [17]

associa ao *site* que pediu sua criação. Após essa fase de configuração inicial, todas as requisições que forem feitas ao servidor serão enviadas com o valor do *Cookie* gravado e, ao receber tais informações, o servidor poderá identificar o cliente e tomar decisões apropriadas.

4.4 Protocolo de correio eletrônico

Além do HTTP, existem outros protocolos importantes utilizados na internet e que podem agregar serviços à estrutura de um sítio. Um deles é o SMTP (*Simple Mail Transfer Protocol*), definido na RFC 5321 [16] e utilizado para comunicação entre servidores de correio eletrônico.

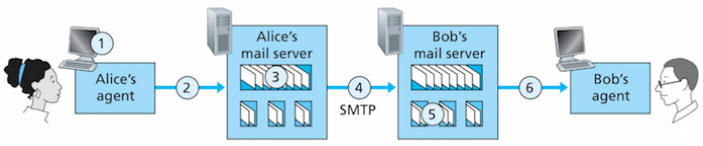


Figura 4.6: Exemplo de interação SMTP. [17]

O SMTP é um protocolo, principalmente, de *PUSH*, ou em outras palavras, a conexão é iniciada pelo lado que envia a mensagem ou que publica. Quando um novo *e-mail* chega no servidor de *e-mail* do destinatário, o servidor já não consegue dar *PUSH* para o computador do usuário por diversas razões. Nesse contexto, para ler suas mensagens, um usuário deve utilizar outro protocolo (POP3, IMAP, HTTP, etc) e, provavelmente, um programa cliente que abstraia isso, que consiga fazer requisições para o servidor que possui sua caixa de correspondência. Esse processo é ilustrado na Figura 4.6.

Enquanto na RFC 5321 o transporte é discutido em cima do TCP, outros transportes são possíveis.

Após autenticação do usuário, as transações de mensagens são feitas em três passos. São eles [16]:

1. A transação começa com o comando **MAIL** que identifica o remetente.
2. Um ou uma série de comandos **RCPT** identificam os destinatários.
3. Um comando **DATA** inicia a transferência da mensagem e termina com um indicador de fim de *e-mail*. Nesse passo, tanto o assunto da mensagem e seu corpo são especificados.

O exemplo a seguir ilustra o envio de um *e-mail* utilizando o protocolo SMTP. Adaptado de [5].

```
$ telnet mail.domain.ext 25
Trying ???.???.???.???...
Connected to mail.domain.ext.
Escape character is '^]'.
220 mail.domain.ext ESMTP Sendmail ?version-number?; ?date+time+gmtoffset?
$ HELO local.domain.name
250 mail.domain.ext Hello local.domain.name [loc.al.i.p], pleased to meet you
$ MAIL FROM: mail@domain.ext
250 2.1.0 mail@domain.ext... Sender ok
$ RCPT TO: mail@otherdomain.ext
250 2.1.0 mail@otherdomain.ext... Recipient ok
$ DATA
354 Send data now. Terminate with a . on a line alone
$ Subject: [assunto]
$
$ [mensagem]
$
```

```
$ .
250 2.0.0 ???????? Message accepted for delivery
$ QUIT
221 2.0.0 mail.domain.ext closing connection
Connection closed by foreign host.
```

4.5 Desenvolvimento

4.5.1 Introdução

Uma ferramenta muito conhecida para o desenvolvimento de aplicações Web é um *framework Web open-source* chamado de Ruby on Rails, ou RoR [4]. Sua arquitetura é MVC (*Model-View-Controller*) e trabalha muito bem com aplicações orientadas a bancos de dados (além de suportar uma grande variedade dos mesmos, como MySQL, Oracle, MS SQL Server, PostgreSQL, IBM DB2, entre outros). O *framework* já possui uma comunidade gigantesca e madura e sítios grandes como, por exemplo, Twitter, GitHub e BaseCamp já a utilizam ou utilizaram.

A popularidade da ferramenta se deve ao fato de que a mesma favorece e incentiva o uso de convenções. Essas convenções são consideradas melhores práticas para realizar algumas tarefas e eliminam, assim, códigos de configuração e aumentando a produtividade. Essas vantagens são originárias do fato de que muitas tarefas comuns ao fazer um sítio já são implementadas nativamente pelo *framework* como, por exemplo: gerência de *e-mail*, mapeamentos entre objetos e banco de dados, estrutura de arquivos, geração de código, organização e nomeação dos elementos. No momento em que o programador as entende e utiliza, o desenvolvimento torna-se, principalmente, muito mais rápido e menos suscetível a erros, e o código menor, mais manutenível e otimizado. Todo o trabalho de montar o servidor, configurar as rotas, definir padrões, entre outros, é reduzido ou eliminado a poucos comandos ou linhas de código e, assim, o desenvolvedor pode focar seu esforço em apresentar páginas com um *design* melhor, implementar algoritmos mais rápidos ou adicionar novas funcionalidades.

Essa ferramenta é construída em cima da linguagem Ruby, que é interpretada, dinamicamente tipificada, *open-source*, orientada a objetos, possui gerência automática de memória e concebida para ser simples, produtiva e elegante [3].

Algumas características desse *framework* serão abordadas a seguir.

4.5.2 Padrão *Model-View-Controller*

O padrão arquitetural MVC, no Ruby on Rails é usado para aumentar a manutenibilidade da aplicação. O *Model*, ou modelo, trata a lógica de negócio, a *View*, ou visualização, gerencia o modo como as páginas e dados serão mostrados, é a representação visual do modelo e o *Controller*, ou controlador, cuida do fluxo da aplicação. O MVC permite uma separação clara de conceitos facilitando, assim, o desacoplamento e teste dos diferentes módulos. [4]

Model

O modelo possui todas as regras de negócio da aplicação e manipulação de dados. Sua função principal, no *framework*, é gerenciar a interação com a informação no banco de dados e fazer as validações apropriadas. Representa os modelos e seus dados, a associação entre esses modelos, a hierarquia entre modelos, valida modelos antes de serem salvos no banco de dados e dá suporte às operações em banco de dados de uma maneira orientada a objetos. [4]

View

A visualização é a interface com o usuário, é o método de mostrar e coletar informações e dados para o e do usuário. As visualizações podem servir o conteúdo em diferentes formatos como HTML, PDF, XML, RSS e outros. [4]

Controller

O controlador é responsável por orquestrar todo o processo para tratar uma requisição no Ruby on Rails. Pode pegar ou salvar dados de um modelo e usar uma visualização para criar o *output* HTML. Um controlador pode ser pensando como a ponte entre modelos e visualizações. Faz com que os dados do modelo estejam disponíveis para as visualizações para que essas possam disponibilizar esses para o usuário e salva ou atualiza os dados do usuário no modelo. [4]

4.5.3 Fluxo de Execução Simplificado

O diagrama da Figura 4.7 exemplifica a relação entre controlador, visualização e modelo descrita anteriormente. Três novos componentes foram adicionados, são eles: servidor Web, Roteador e Dispatcher. O servidor Web recebe uma requisição vinda de um *browser*, essa requisição é traduzida pelo Roteador que irá dizer qual o controlador correto que deve ser invocado, o Dispatcher carrega o controlador correto e passa o controle para ele. Uma

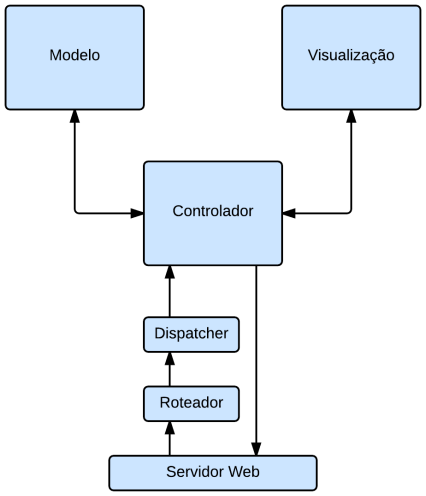


Figura 4.7: Componentes importantes do Ruby on Rails e como eles se relacionam.

vez com o controle, o controlador orquestra a execução da requisição até que, por fim, envie, como resposta, uma visualização.

4.5.4 RubyGems

A linguagem Ruby possui um mecanismo muito interessante de compartilhamento de bibliotecas e resolução de dependências denominado RubyGems. Existem mais de 70.000 [2] Gems e grande parte dessas é voltada para o desenvolvimento Web. A oferta de bibliotecas é das mais variadas, todas são, também, *open-source* e podem resolver problemas, por exemplo, de autenticação, interação com redes sociais, administração do sítio, etc. A vantagem do uso dessas ferramentas pré-construídas é o fato de não ser necessário fazer um trabalho já feito e testado por, na maioria das vezes, um grande número de colaboradores, o que diminui muito a incidência de erros, brechas de segurança e tempo de desenvolvimento de um produto.

4.5.5 *Hello World*

Nesta seção, a criação de uma aplicação simples (*Hello World*) será demonstrada.

```
$ rails new HelloWorld
[...]
```

Abrindo o terminal, o primeiro comando é digitado, dezenas de pastas e arquivos são criados e Gems padrão são colocadas no projeto para dar suporte à construção da aplicação Web. O *output* desse passo é omitido.

```
$ cd ./HelloWorld/
$ rails generate controller home index
      create  app/controllers/home_controller.rb
       route  get "home/index"
      invoke  erb
      create    app/views/home
      create    app/views/home/index.html.erb
      invoke  test_unit
      create    test/controllers/home_controller_test.rb
      invoke  helper
      create    app/helpers/home_helper.rb
      invoke    test_unit
      create      test/helpers/home_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/home.js.coffee
      invoke    scss
      create      app/assets/stylesheets/home.css.scss
```

Tendo criado a base da aplicação, o diretório de trabalho é mudado para a raíz do projeto e o comando para gerar uma primeira página é executado. Neste passo, também, há a criação de mais alguns arquivos e configurações.

```
$ sublime ./config/routes.rb
```

```
1  HelloWorld::Application.routes.draw do
2    get "home/index"
3    root 'home#index'
4  end
```

Figura 4.8: Mudando a página índice da aplicação.

Neste passo, uma nova linha é adicionada às rotas do servidor: `root 'home#index'`, conforme Figura 4.8. Essa linha faz com que a página índice seja a vinculada à ação `index` do controlador `home`.

```
$ sublime ./views/home/index.html.erb
```

Neste passo, o conteúdo da página índice da aplicação é alterado para mostrar "*Hello World*", conforme Figura 4.9.

```
1   <h1>Hello World</h1>
```

Figura 4.9: Mudando o conteúdo da página índice.

```
$ rails server
=> Booting WEBrick
=> Rails 4.0.1 application starting in development on http://0.0.0.0:3000
=> Run ‘rails server -h‘ for more startup options
=> Ctrl-C to shutdown server
[2014-07-10 17:12:25] INFO  WEBrick 1.3.1
[2014-07-10 17:12:25] INFO  ruby 1.9.3 (2013-11-22) [x86_64-darwin13.0.1]
[2014-07-10 17:12:25] INFO  WEBrick::HTTPServer#start: pid=6219 port=3000
```

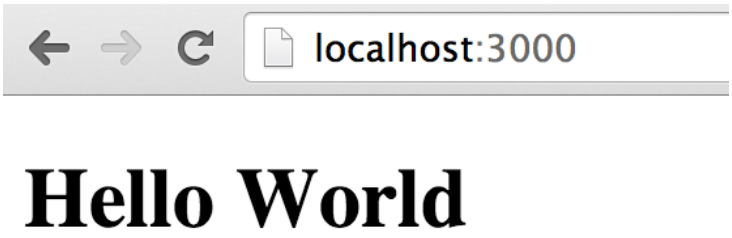


Figura 4.10: Acessando a aplicação.

Finalizando, o servidor é inicializado e disponível para acesso em `http://localhost:3000`, conforme Figura 4.10.

Capítulo 5

Desenvolvimento do Sistema Web

O intuito deste capítulo é apresentar o problema e a respectiva solução projetada.

5.1 Definição do Problema

O problema que este projeto visa resolver é o de tornar o CUDAlign (Capítulo 3) não só mais conhecido mas também mais acessível para usuários de qualquer parte do mundo com acesso à *internet*. Visa também habilitar tanto o usuário experiente quanto o inexperiente para usar a ferramenta CUDAlign para obter o alinhamento ótimo entre duas sequências de DNA.

5.2 Escopo da Solução

Tendo o problema da Seção 5.1 em mãos, foram levantados alguns requisitos que a solução deveria atender para que fosse considerada bem-sucedida, ou seja, deveria ser capaz de:

- Disponibilizar uma interface gráfica amigável para execução do CUDAlign;
- Obter as sequência a serem comparadas via um sítio na internet e carregá-las na máquina na qual o CUDAlign será executado;
- Receber os parâmetros fornecidos pelo usuário via Web, montar o *script* de execução do CUDAlign e executá-lo na GPU apropriada;
- Projetar um meio de monitoramento do andamento da execução;
- Enviar os dados para o usuário quando a execução terminar.

5.3 Visão Geral

A Figura 5.1 mostra a visão geral da execução da solução proposta bem como a divisão de competências entre o processo Interno e Externo. Os quadrados pintados em amarelo indicam componentes do processo Interno e os quadros pintados em verde indicam componentes do processo Externo. Essa figura e o próprio processo que ela explica podem ser vistos conforme a seguinte sequência de passos:

1. O usuário, via protocolo HTTP (Seção 4.3), faz uma requisição para que o alinhamento entre duas sequências seja calculado. Os arquivos de sequências podem ser enviados pelo próprio usuário ou o trabalho de aquisição pode ser passado para a solução. Essa requisição é recebida, validada e salva em um banco de dados.
2. Caso a requisição salva no banco de dados não tenha *downloads* pendentes (não necessite adquirir as sequências por si só), o passo 3 (próximo passo) é pulado.
3. O *Accession ID*, ou o número que identifica uma determinada sequência biológica no NCBI [11], é utilizado para adquirir a sequência no próprio repositório do NCBI.
4. O CUDAlign é executado com os parâmetros vindos de requisições sem pendências e, então, monitorado. O CUDAlign fornece muitos dados sobre sua execução e fornece, por fim, dados sobre o resultado do alinhamento.
5. Os dados de execução e resultado são gravados no banco.
6. O usuário é comunicado que o processamento de sua requisição de alinhamento acabou via *email*.
7. O usuário, por fim, solicita, via HTTP, o resultado do alinhamento. Esse resultado pode ser um resumo do alinhamento ou, ainda, se o usuário desejar, o arquivo de saída do CUDAlign na íntegra.

5.4 Estruturação da Solução

Para melhor estruturar a solução e facilitar sua manutenção, foram projetados dois processos denominados Externo e Interno. O processo Externo tem como dever cuidar de toda parte relacionada à interação com o usuário. O processo Interno, por outro lado, suporta e gerencia a execução do algoritmo CUDAlign.

Os processos citados acima não se comunicam diretamente e dependem de um banco de dados para isso. A Figura 5.2 mostra as interações entre esses três módulos da solução.

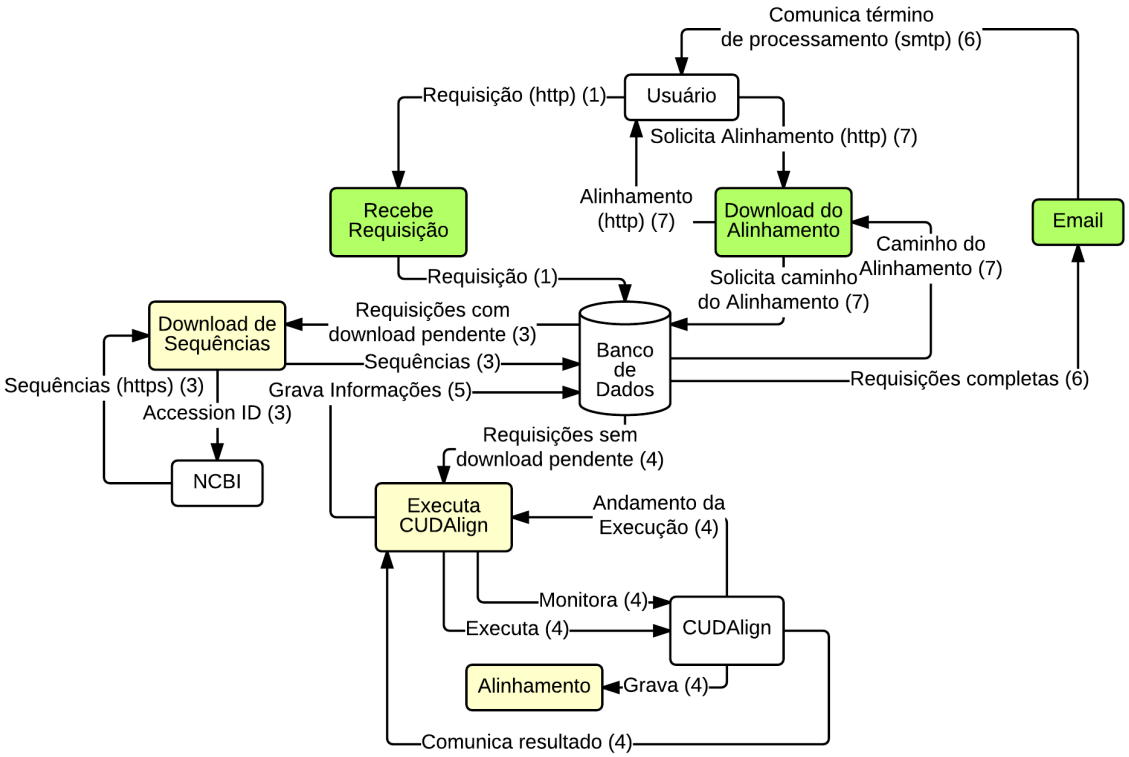


Figura 5.1: Visão geral da execução da solução em conjunto com o CUDAlign.

5.5 Componentes da Solução

5.5.1 Banco de Dados

O banco de dados é o canal de comunicação entre os processos Interno e Externo. As informações trocadas por eles são para coordenar suas ações, para controlar e analisar a execução do algoritmo CUDAlign e, por fim, para gerar um resultado para o usuário requisitante.

O canal de comunicação citado foi criado utilizando o banco de dados MySQL [22].

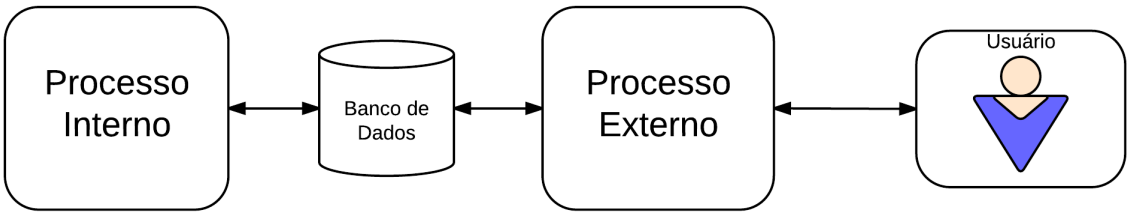


Figura 5.2: Componentes da solução e interação com o usuário.

Este banco foi escolhido por ser *open source*, por apresentar ferramentas satisfatórias de administração, por atender o projeto em sua plenitude e por apresentar conectores de fácil acesso para a linguagem Java.

Este banco de dados foi idealizado com apenas uma tabela de 38 colunas. Essas colunas guardam as mais diversas informações de controle e administração dos processos Interno e Externo e informações de controle e execução do CUDAlign. Todas as colunas são importantes para a execução correta de toda a solução mas uma em especial é responsável pela coordenação dos processos: a coluna *status*. A coluna *status* identifica em qual estado, ou seja, qual a situação atual da requisição de alinhamento. Esta coluna pode assumir 9 valores diferentes, um único valor por vez, são eles:

- Não Iniciada: indica que a requisição de alinhamento ainda não foi processada
- Fila de *Downloads*: indica que a requisição de alinhamento precisa que seja feito o *download* de uma ou de ambas sequências
- *Download* de Sequências Falhou: indica que o processo Interno falhou ao tentar fazer o *download* de uma sequência
- Fazendo *Download* de Sequências: indica que o processo Interno está fazendo o *download* de uma ou de ambas sequências
- Pronta: indica que a requisição de alinhamento cumpriu todos os pré-requisitos e está na fila para ser executada
- Calculando Alinhamento: indica que o alinhamento está sendo calculado
- Falhou ao Calcular Alinhamento: indica que ocorreu alguma falha quando o processo Interno estava tentando calcular o alinhamento entre as duas sequências
- Completa: indica que todos os passos para calcular o alinhamento foram cumpridos com sucesso
- Sequência Não Encontrada: indica que o processo Interno não conseguiu encontrar uma ou ambas sequências

Mais uma vez é importante frisar o papel dos estados de alinhamento: eles coordenam a execução de toda a solução.

5.5.2 Processo Externo

O papel do processo Externo é fornecer uma interface gráfica com o usuário tanto de *input* quanto de *output*. Essa interface serve para informar sobre o CUDAlign, ou seja,

o que faz, como faz, o que são e quais são os parâmetros de sua execução e autores, coletar parâmetros de execução do algoritmo (incluindo informações sobre a sequência ou a própria sequência) e validá-los, mostrar dados sobre a execução do algoritmo, mostrar o resultado da execução do algoritmo e disponibilizar o resultado, se calculado com sucesso, do alinhamento entre duas sequências biológicas.

Visão Geral da Execução do Processo Externo

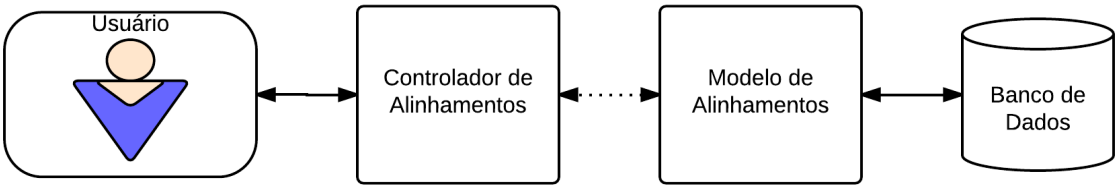


Figura 5.3: Diagrama de execução do processo Interno de forma simplificada.

A Figura 5.3 mostra a visão geral de execução do processo Externo: tendo um usuário enviado uma requisição para o sítio que hospeda a solução, o método correto é chamado no Controlador de alinhamentos. O Controlador, por sua vez, dá o controle ao Modelo de alinhamentos quantas vezes precisar (para criar, recuperar, atualizar, deletar ou validar objetos de alinhamento). O Modelo de alinhamentos tendo terminado sua tarefa, devolve o controle ao Controlador que, por fim, serve a Visualização para o usuário que fez a requisição.

Implementação

O processo Externo foi implementado usando a linguagem Ruby e o *framework* Ruby on Rails (Seção 4.5). Neste projeto, foram implementados um Modelo, um Controlador e algumas Visualizações.

O Modelo de alinhamentos do processo Externo é responsável por, além de seu propósito geral discutido na Seção 4.5.2, validar a entrada de parâmetros para execução do CUDAlign quanto à presença e formato, deixar os dados tanto de entrada do algoritmo quanto de configuração prontos para serem lidos pelo processo Interno.

Coordenação com o Processo Interno

O processo Externo processa requisições de alinhamentos com os seguintes estados (Seção 5.5.1) e da seguinte maneira:

- Não Iniciada: ao incluir uma requisição de alinhamento no banco, o processo Externo a inclui como "Não Iniciada"
- *Download* de Sequências Falhou, Falhou ao Calcular Alinhamento, Sequência Não Encontrada e Completa: utilizados para mostrar na página de acompanhamento da requisição de alinhamento uma formatação diferenciada e para enviar *e-mails* para os usuários requisitantes comunicando a falha ou sucesso do cálculo
- Os estados restantes não são processados de qualquer maneira pelo processo Externo

5.5.3 Processo Interno

O processo Interno tem como responsabilidade dar suporte para a execução do algoritmo CUDAlign e salvar informações sobre essa execução para que sejam repassadas ao usuário pelo processo Externo.

Implementação, Coordenação e Visão Geral de Execução do processo Interno

O processo Interno foi todo implementado utilizando a linguagem Java. Foram implementados quatro componentes principais, são eles o Distribuidor, o Gerenciador de *Downloads*, o Gerenciador de Processos e a Camada de Acesso ao Banco de Dados.

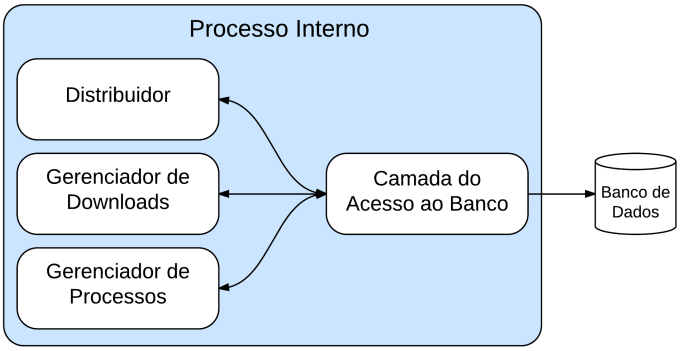


Figura 5.4: Componentes do processo Interno.

A Figura 5.4 mostra os componentes do processo Interno e como estes se comunicam. Pelo diagrama, é possível perceber que o Distribuidor, o Gerenciador de *Downloads* e o Gerenciador de Processos, que são três *threads* separadas, não se comunicam diretamente mas usam a Camada de Acesso ao Banco de Dados para isso. A Camada de Acesso ao Banco de Dados, por sua vez, se comunica com o banco de dados.

Distribuidor

O Distribuidor é uma *thread* que acessa o banco de dados de tempos em tempos

para procurar atualizações em requisições de alinhamento marcadas com o estado "Não Iniciada". Encontradas novas requisições de alinhamento, processa-as para determinar se estão válidas, se necessitam de pré-requisitos (como o *download* de sequências) ou se já estão prontas para serem processadas pelo CUDAlign. Saindo do Distribuidor, uma requisição de alinhamento de sequências pode assumir os estados "Pronta", "Fila de *Downloads*"ou "Sequência Não Encontrada".

Gerenciador de Downloads

O Gerenciador de *Downloads* é uma *thread* que acessa o banco de dados de tempos em tempos para procurar atualizações em requisições de alinhamento marcadas com o estado "Fila de *Downloads*"e "Fazendo *Download* de Sequências". O último estado é interessante para ter alguma proteção contra falhas. Enquanto o *download* de uma sequência está sendo feito, é atribuída à requisição o estado "Fazendo *Download* de Sequências". Os estados de saída de uma requisição de alinhamento do Gerenciador de *Downloads* são "Pronta"e "*Download* de Sequências Falhou".

O Gerenciador de *Downloads* monitora os *downloads*, monitora quantos estão na fila, monitora quantos estão sendo processados no momento, diz onde o *download*, uma vez completo, será salvo, impõe quantos *downloads* concorrentes podem ser feitos, determina o número de tentativas de *download* de uma determinada sequência, monitora *downloads* já realizados para que, se possível, não seja necessário fazê-los novamente, entre outros.

Gerenciador de Downloads: Tolerância a Falhas

Além disso, esse Gerenciador possui dois mecanismos de tolerância a falhas. O primeiro verifica, ao iniciar a *thread*, todos os *downloads* que estão com o estado "Fazendo *Download* de Sequências". Esse estado é um forte indicativo que a *thread* foi finalizada sem concluir *downloads* pendentes. Por esse motivo, todos os que estão com esse estado no momento de inicialização da *thread* são colocados com prioridade na fila, o conteúdo que foi feito o *download* para essa requisição é apagado e o *download* é reinicializado. O segundo mecanismo é o número de tentativas e o espaço de tempo entre essas tentativa. É um mecanismo interessante dado a rede muito congestionada ou ausência temporária dela.

Gerenciador de Processos

Gerenciador de Processos é a terceira *thread* que procura, no banco de dados, requisições de alinhamentos prontas para serem executadas, ou seja, com o estado "Pronta". O estado intermediário que uma requisição assume, neste componente, é o de "Calculando Alinhamento"e pode, na saída, assumir os estados "Falha ao Calcular Alinhamento"e "Completa".

O Gerenciador é responsável por validar a saída de execução do algoritmo, salvar certas linhas do *output* em *console* do CUDAlign no banco, limpar arquivos desnecessários de saída do CUDAlign caso a execução tenha sido realizada com sucesso, salvar no banco o caminho de arquivos importantes de resultado do alinhamento para serem lidos ou para que seu *download* seja feito pelo usuário, salvar *previews* e o tamanho do arquivo de resultado para mostrar ao usuário antes do *download* ser efetuado, gravar no banco informações analíticas como, por exemplo, hora que começou o cálculo, hora que terminou e tempo estimado de execução e fornecer uma certa tolerância a falhas.

Gerenciador de Processos: Tolerância a Falhas O Gerenciador de Processos conta com um mecanismo de tolerância a falhas parecido com o empregado ao Gerenciador de *Downloads*. Ao iniciar sua execução, a *thread* do Gerenciador de Processos busca por processos com o estado "Calculando Alinhamento"a fim de encontrar aquelas que, por algum motivo, não tiveram sua execução concluída por falhas não planejadas. Graças aos mecanismos de tolerância a falhas do próprio CUDAlign (Seção 3.2.3), a execução do algoritmo pode ser retomada do ponto em que parou. Além disso, para tentar identificar a falta que deu origem à falha, se possível, o salvamento das linhas de *output* em *console* podem ajudar.

Camada de Acesso ao Banco de Dados

A Camada de Acesso ao Banco de Dados foi pensada para dar suporte às inúmeras escolhas de bancos de dados que o Ruby on Rails suporta (Seção 4.5.1). Desse modo, em caso de necessidade, será possível, sem muito esforço, migrar a aplicação para utilizar outro banco que não o MySQL. Outra razão para essa camada existir é que a manutenção de código fica confinada em apenas um local, ou seja, a manutenção é feita em um ponto único com esforço reduzido.

5.5.4 Customização

Um certo poder de customização foi introduzido no projeto para que pequenas configurações corriqueiras não levassem um usuário a programar. Locais de salvamento de todos os arquivos, nomes dos arquivos, nome das tabelas no banco de dados, URL do banco de dados, *driver* do banco de dados, usuários e senhas do banco de dados, apontadores de resultado do cálculo do alinhamento, tempo de espera para as diversas *threads* consultarem o banco, tempo entre tentativas de *download*, número máximo de tentativas de *download*, linhas de *log* que serão salvas no banco, número de linhas do resultado a serem salvas no banco e número máximo de *downloads* concorrentes são algumas das variáveis que podem

ser modificadas para otimizar a execução do processo Interno. Quanto ao processo Externo, grande parte por ter sido implementado utilizando uma linguagem interpretada, além de configurações que podem ser alteradas *on-the-fly* é possível, também, alterar a lógica de negócio, o controlador e as visualizações.

5.6 Fluxo de Execução

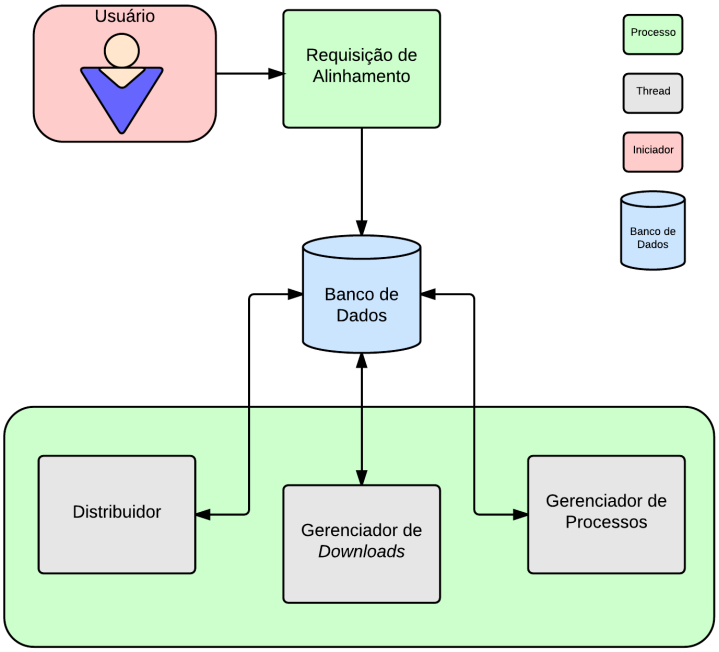


Figura 5.5: Visão geral dos componentes da solução.

Pelo diagrama da Figura 5.5, é possível ter uma visão geral da execução do processo Interno e Externo quando há uma requisição de alinhamento entre duas sequências biológicas. Apesar dos processos e *threads* dos mesmos estarem sempre executando, no sentido de que toda hora consultam o banco para buscar atualizações, o iniciador da execução, de um ponto de vista lógico, é o usuário.

O processo Interno é composto pela parte de requisição de alinhamento. O processo Externo é composto por três *threads* independentes entre si que se comunicam estritamente pelo banco de dados. A comunicação processo Interno-Externo é feita, também, estritamente pelo banco de dados.

No diagrama da Figura 5.6, fica evidente o caminho lógico percorrido por uma requisição de alinhamento de forma simplificada:

1. O usuário inicia todo o processo da solução requisitando um novo alinhamento.

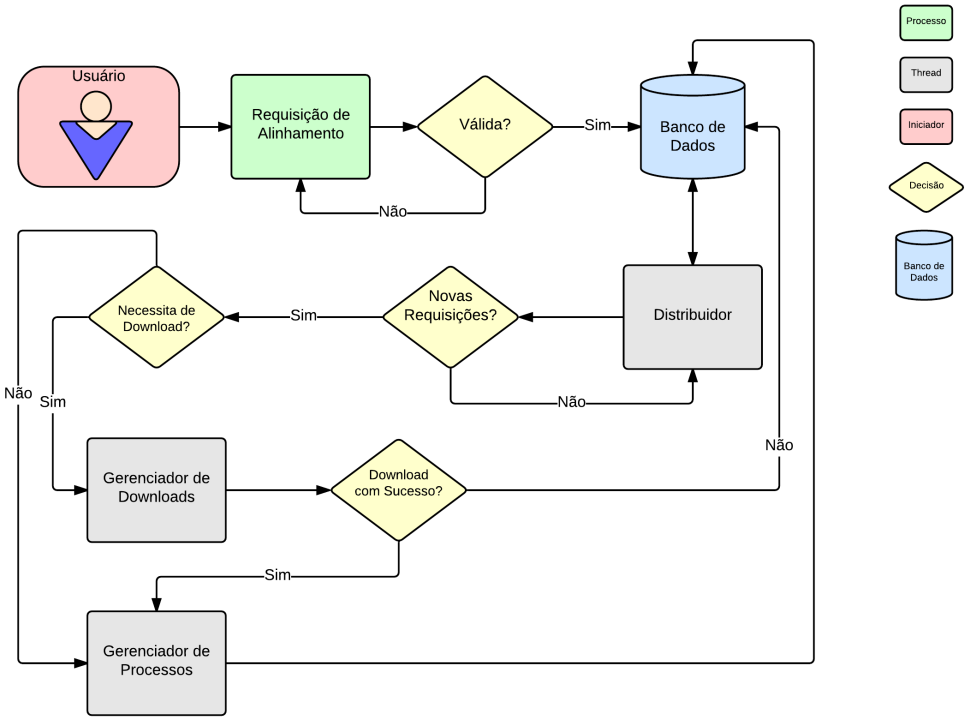


Figura 5.6: Caminho lógico da requisição de alinhamento.

2. O processo Externo, que cuida da requisição deste novo alinhamento, verifica a validade da requisição. Se for válida, salva a mesma no banco de dados ou, se não for, volta para a etapa de requisição.

3. O processo Interno, mais precisamente a *thread* do Distribuidor, detecta a atualização no banco de dados e verifica a necessidade de *download*. Se o *download* de uma ou de ambas sequências for necessário, o controle sobre a requisição de alinhamento passa a ser do Gerenciador de *Downloads*. Se não for necessário fazer o *download* de nenhuma sequência, o algoritmo vai para o passo 5.

4. Com o controle do alinhamento, o Gerenciador de *Downloads* tenta fazer o *download* da(s) sequência(s). Realizado com sucesso, o controle da requisição passa para o Gerenciador de Processos. Realizado com falha, o algoritmo não consegue prosseguir e comunica, via banco de dados, a falha que ocorreu neste passo.

5. Estando com o controle do alinhamento, o Gerenciador de Processos executa o CUDAlign. Sendo executado com sucesso ou com erro, o processo Interno comunica, via banco de dados, o processo Externo.

6. O processo Externo, por fim, mostra os dados para o usuário final.

Capítulo 6

Resultados

O capítulo de Resultados visa mostrar o resultado final sob a ótica do usuário final, o público alvo da solução apresentada no Capítulo 5.

6.1 Introdução

A ideia deste capítulo é mostrar todo o fluxo de execução visto no Capítulo 5 sob a ótica de um utilizador final.

O sistema Web proposto no presente trabalho de graduação foi testado em um *desktop* com processador Intel i5-4570 3.6 GHz, memória de 16GB RAM, placa de vídeo Nvidia GTX 770 e conectado à *internet*.

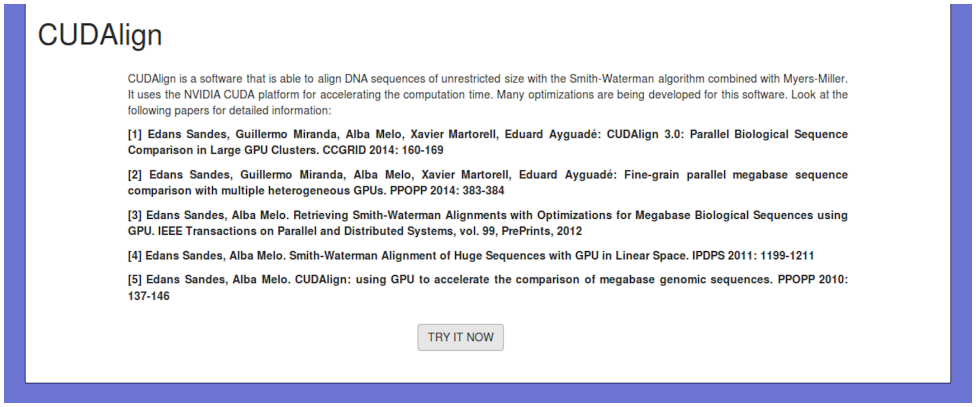


Figura 6.1: Página inicial da solução

6.2 Página Índice

A página índice, ou inicial, da solução apresenta, como pode ser visto na Figura 6.1, um breve resumo do que é o CUDAlign e algumas referências caso o usuário final tenha o

desejo de aprofundar seus conhecimentos no algoritmo.

Embaixo das referências bibliográficas há um botão que convida o usuário a testar o CUDAlign. Quando o usuário seleciona esse botão, ele é redirecionado para a página de requisição de um novo alinhamento.

6.3 Página de Requisição de Novo Alinhamento

A página de requisição de novo alinhamento pode ser vista na Figura 6.2. O objetivo principal dessa página é fornecer ao usuário uma interface amigável para passagem de parâmetros relevantes para execução da ferramenta CUDAlign.

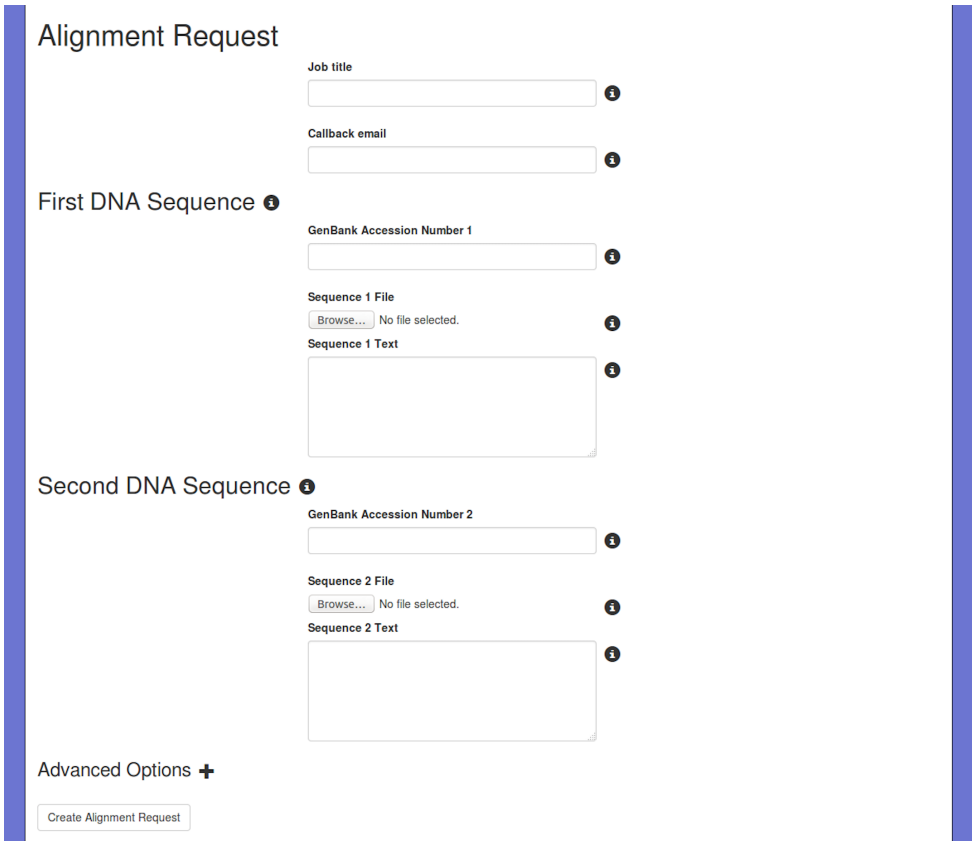


Figura 6.2: Página de Requisição de Novo Alinhamento e parâmetros obrigatórios

Para requisitar um novo alinhamento, um usuário deve prover, no mínimo, um nome descritivo para o alinhamento, um *email* e as sequências. Essas sequências podem ser referenciadas por um *Accession ID* (Seção 5.3), providas a partir de um arquivo do próprio usuário ou até mesmo coladas no campo de texto livre. O usuário pode, ainda, modificar alguns parâmetros mais avançados do CUDAlign (Figura 6.3).

Para facilitar o entendimento do usuário sobre os parâmetros, foram colocadas explicações ao lado de cada caixa do formulário de entrada de dados, como na Figura 6.4.

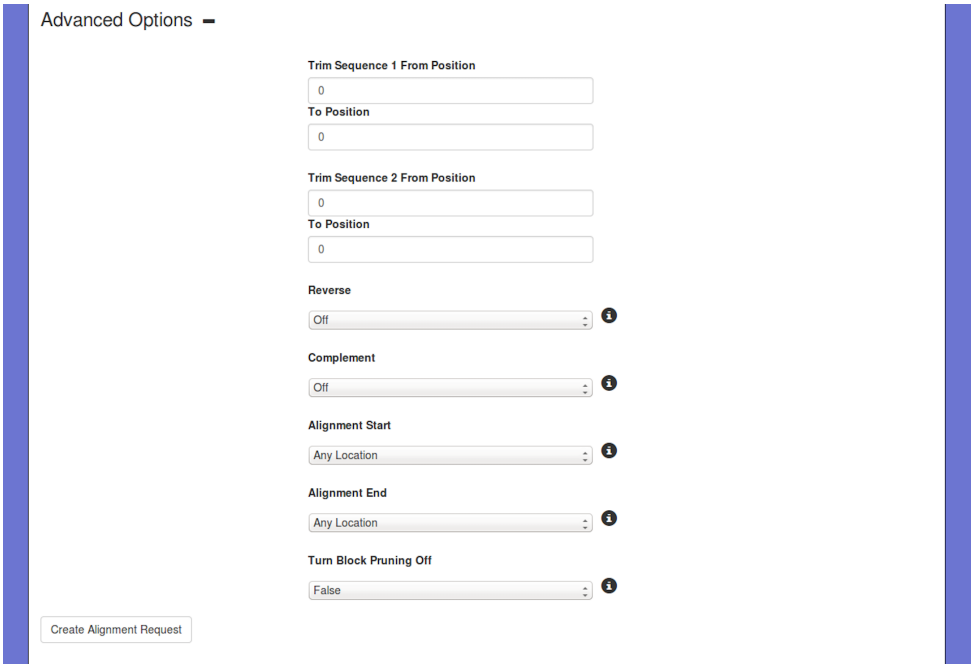


Figura 6.3: Opções avançadas de requisição de novo alinhamento

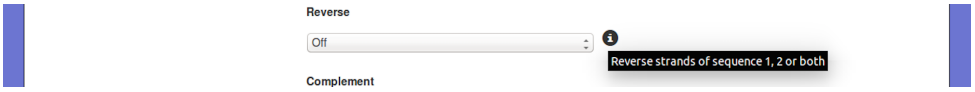


Figura 6.4: Exemplo de explicação de um parâmetro no formulário de entrada de dados

Além disso, validações na entrada de dados também ajudam a explicar alguns parâmetros e previnir que dados incorretos sejam salvos no banco de dados (Figura 6.5).

Após submissão correta do formulário de entrada de dados, como na Figura 6.6, o usuário é redirecionado para a página de acompanhamento da requisição de alinhamento.

6.4 Página de Acompanhamento da Requisição de Alinhamento

A página de acompanhamento da requisição de alinhamento foi construída para que o usuário final pudesse acompanhar a execução da solução e da ferramenta CUDAlign. Como é possível ver na Figura 6.7, a página é muito parecida com descrita na Seção 6.3. Essa página possui, além dos parâmetros de execução que foram usados, o estado em que se encontra a requisição de alinhamento (Seção 5.5.1).

Ao passo que o processo Interno (Seção 5.5.3) executa, a requisição de alinhamento assume diversos estados. Essa mudança de estado, para o utilizador final, pode ser vista nas Figuras 6.8, 6.9, 6.10, 6.11, 6.12, 6.13 e 6.14.

Figura 6.5: Validações e mensagens de erro após submissão do formulário de entrada de dados

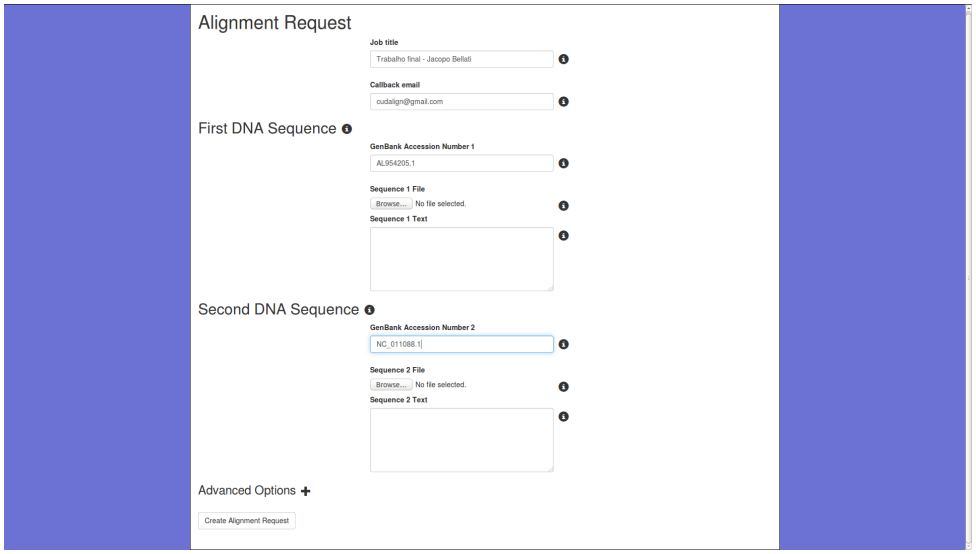


Figura 6.6: Exemplo de preenchimento do formulário de maneira correta

As Figuras 6.12 e 6.13 mostram um novo botão. Esse aparece somente quando o CUDAlign está sendo ou foi executado. O botão serve para acessar o *log* do CUDAlign. No caso do estado "Calculando Alinhamento", é possível acompanhar a execução do CUDAlign em tempo real. No caso do estado "Falhou ao Calcular Alinhamento", esse acesso é interessante pois possibilita a identificação da causa da falha já que a página de monitoramento irá mostrar as últimas linhas salvas do *log*. No caso da Figura 6.14, cuja requisição está "Completa", além da possibilidade de visitar as últimas linhas do *log*, é possível, também, ir à página de resultados ou fazer o *download* do resultado do alinhamento (Figura 6.15).

6.5 Página de Monitoramento do CUDAlign

A página de monitoramento do CUDAlign é a página que mostra informações sobre o estado atual da requisição de alinhamento, a hora que a ferramenta CUDAlign começou sua execução, a hora que acabou sua execução (caso já tenha acabado) e suas últimas linhas de *output* em *console*. Quando executando, as últimas linhas do *log* são atualizadas

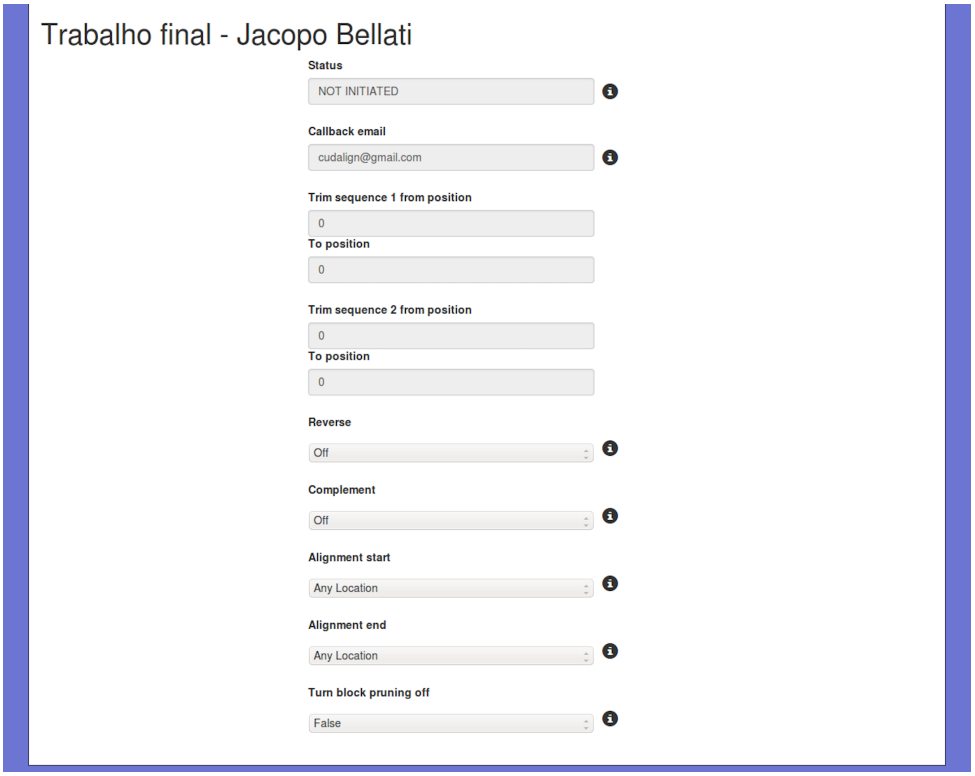


Figura 6.7: Página de visualização de dados relacionados à requisição de alinhamento

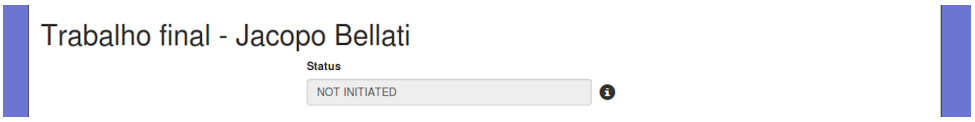


Figura 6.8: Requisição de alinhamento com estado 'Não Iniciada'

constantemente o que permite um monitoramento em tempo real. A Figura 6.16 mostra a página de monitoramento de uma requisição com estado "Completa".

6.6 Envio de *Email* ao Usuário

Assim que o CUDAlign termina sua execução, seja com sucesso ou erro, um *email* é enviado ao usuário informando que o resultado está disponível para ser consultado, como mostrado na Figura 6.17.

6.7 Página de Resultado do CUDAlign

A página de resultado do CUDAlign, que pode ser vista na Figura 6.18, mostra informações sobre o arquivo de saída (alinhamento) da execução da ferramenta. Nessa página, além da possibilidade de efetuar o *download* do alinhamento como na Seção 6.4, infor-

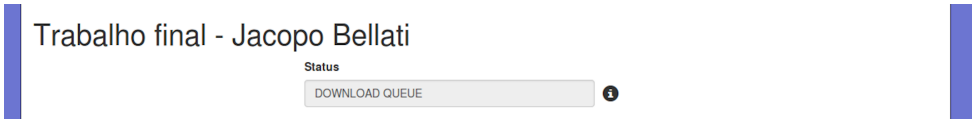


Figura 6.9: Requisição de alinhamento com estado 'Fila de *Downloads*'

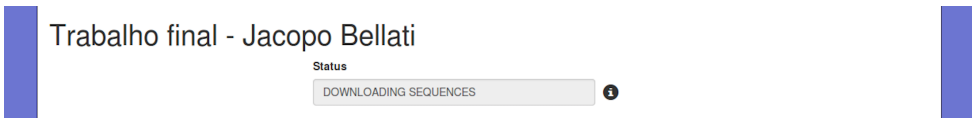


Figura 6.10: Requisição de alinhamento com estado 'Fazendo *Download* de Sequências'

mações relevantes sobre o resumo do *output* podem ser vistas, como tamanho do arquivo de saída, início do arquivo de saída (que contém informações gerais sobre as sequências comparadas) e final do arquivo de saída (que contém um resumo do alinhamento).

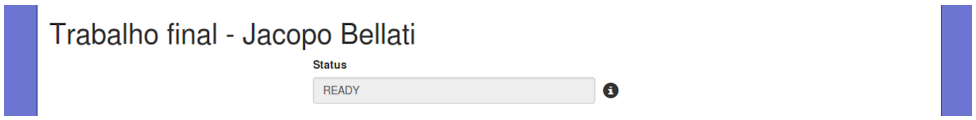


Figura 6.11: Requisição de alinhamento com estado ‘Pronta’

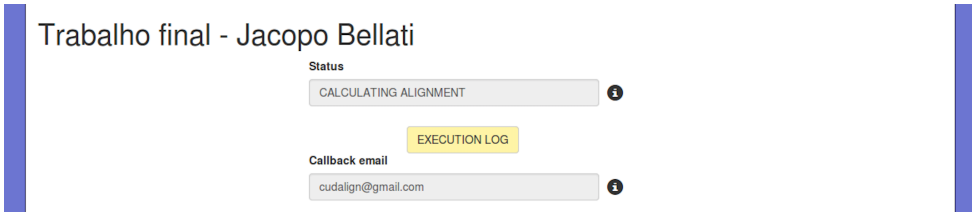


Figura 6.12: Requisição de alinhamento com estado ‘Calculando Alinhamento’

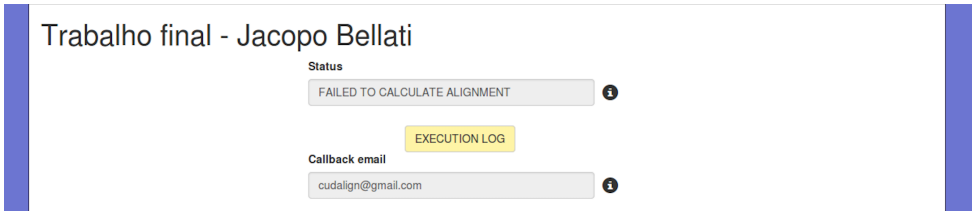


Figura 6.13: Requisição de alinhamento com estado ‘Falhou ao Calcular Alinhamento’

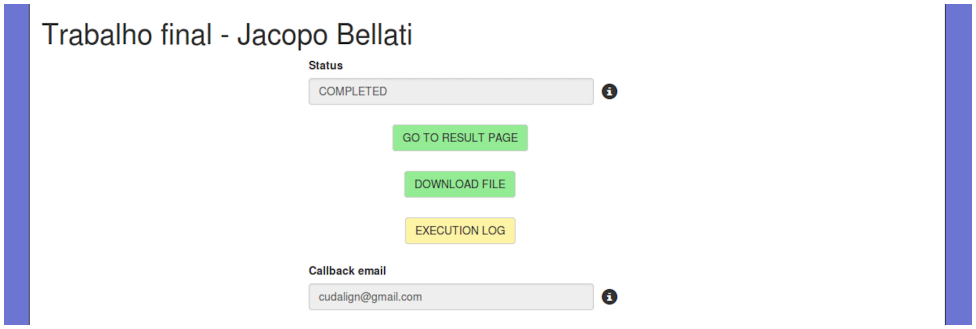


Figura 6.14: Requisição de alinhamento com estado ‘Completa’

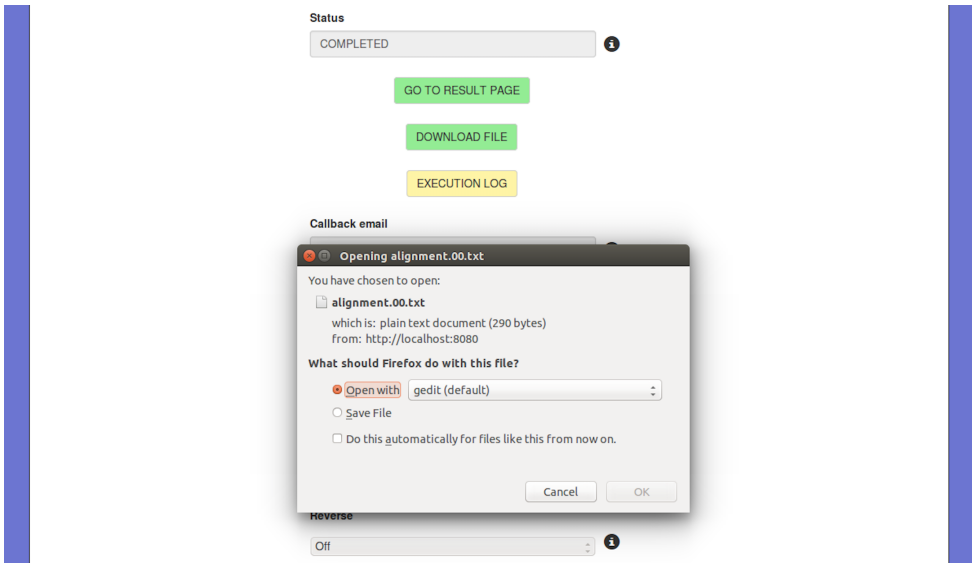


Figura 6.15: *Download* do resultado do alinhamento entre duas sequências

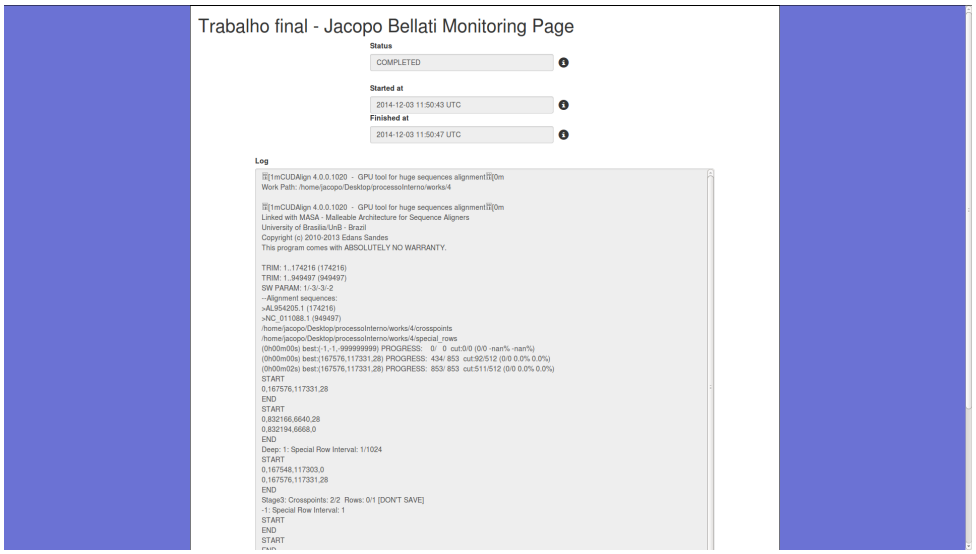


Figura 6.16: Página de monitoramento do CUDAlign

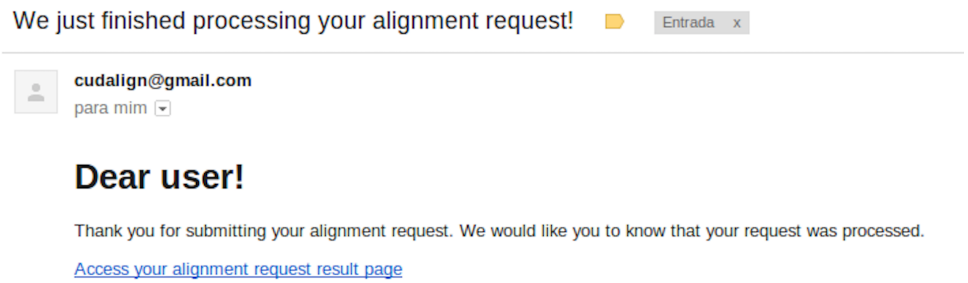


Figura 6.17: *Email* enviado para o usuário após alinhamento acabar de ser processado pelo CUDAlign

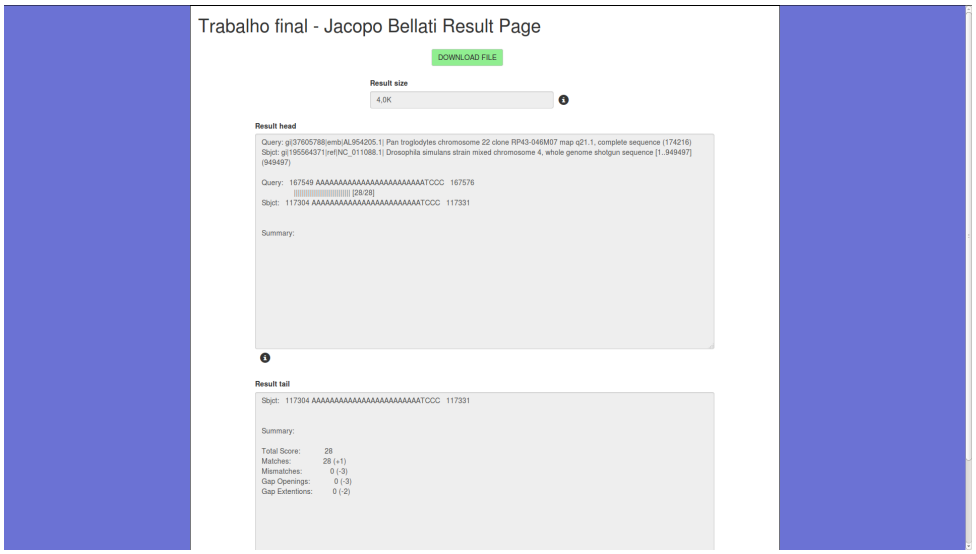


Figura 6.18: Página de resultado do CUDAlign

Capítulo 7

Conclusões

O presente trabalho de graduação propôs uma interface de uso para tornar o algoritmo CUDAlign mais atrativo para usuários finais. Implementada com o intuito de remover parte da complexidade envolvida em termos de instalação, configuração e manuseio do CUDAlign, bem como de obtenção da infraestrutura necessária, a ferramenta proposta atingiu os requisitos propostos para ser considerada bem-sucedida e pode, agora, servir requisições de alinhamento entre duas sequências biológicas de forma simples para usuários de diferentes *backgrounds*.

Como trabalhos futuros, primeiramente sugere-se fazer melhorias e adicionar funcionalidades à ferramenta proposta. Uma funcionalidade interessante seria adicionar suporte ao cálculo de mais de um alinhamento em paralelo, sendo esse paralelismo feito por duas ou mais GPUs em uma mesma máquina ou em diferentes máquinas. Em segundo lugar, uma adição interessante à comunidade científica seria tornar esta ferramenta um *framework* para pessoas que desejassem prover uma interface gráfica amigável para seu algoritmo ou, ainda, disponibilizá-lo na Web pudessem fazê-lo sem muito esforço de desenvolvimento e integração.

Referências

[1] Internet engineering task force, Julho 2014. 23

[2] Ruby gems, Julho 2014. 36

[3] Ruby language, Julho 2014. 34

[4] Ruby on rails, Julho 2014. 34, 35

[5] Telnet - smtp commands (sending mail using telnet), Julho 2014. 33

[6] Edans F. de O. Sandes, Guillermo Miranda, Xavier Martorell, Eduard Ayguade, George Teodoro, and Alba C. M. A. de Melo. Masa: a multi-platform architecture for sequence aligners with block pruning. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6, 2007. x, 8, 9

[7] Edans Flavius de O. Sandes and Alba Cristina M.A. de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. *Parallel and Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199–1211, 2011. 21

[8] Edans Flavius de O. Sandes and Alba Cristina M.A. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 24:1009–1021, 2013. x, 1, 2, 15, 17, 19, 20

[9] Edans Flávius de Oliveira Sandes. Comparação paralela de sequências biológicas longas utilizando unidades de processamento gráfico (gpus). Master's thesis, Universidade de Brasília, 2011. x, 11, 13, 14, 15, 16, 17, 18, 19

[10] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998. x, 1, 8

[11] National Center for Biotechnology Information. National center for biotechnology information, November 2014. 40

[12] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162:705–708, 1982. 1, 9

[13] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18:341–343, 1975. 10

[14] IETF. Rfc 1945, Julho 2014. 25, 28

[15] IETF. Rfc 2616, Julho 2014. 25, 27, 28, 30

[16] IETF. Rfc 5321, Julho 2014. 32, 33

[17] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach, 6th Edition*. Addison Wesley, 2012. x, 24, 25, 26, 28, 30, 31, 32

[18] David Mount. *Bioinformatics: Sequence and Genome Analysis, Second Edition*. Cold Spring Harbor Laboratory Press, 2004. 6

[19] E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988. 1, 10

[20] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48:443–453, 1970. 1, 7

[21] NVidia. Cuda, Julho 2014. 12

[22] Oracle. Mysql, November 2014. 41

[23] J.D. Owens, D. Luebke M. Houston, J.E. Stone S. Green, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96:879–899, 2008. 1

[24] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147:195–197, 1981. 1, 8

[25] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17:419–470, 1985. x, 24, 25